1 A new design for the Web

This chapter covers

- Asynchronous network interactions and usage patterns
- The key differences between Ajax and classic web applications
- The four fundamental principles of Ajax
- Ajax in the real world

Ideally, a user interface (UI) will be invisible to users, providing them with the options they need when they need them but otherwise staying out of their way, leaving users free to focus on the problem at hand. Unfortunately, this is a very hard thing to get right, and we become accustomed, or resigned, to working with suboptimal UIs on a daily basis—until someone shows us a better way, and we realize how frustrating our current method of doing things can be.

The Internet is currently undergoing such a realization, as the basic web browser technologies used to display document content have been pushed beyond the limits of what they can sanely accomplish.

Ajax (Asynchronous JavaScript + XML) is a relatively recent name, coined by Jesse James Garrett of Adaptive Path. Some parts of Ajax have been previously described as *Dynamic HTML* and *remote scripting*. Ajax is a snappier name, evoking images of cleaning powder, Dutch football teams, and Greek heroes suffering the throes of madness.

It's more than just a name, though. There is plenty of excitement surrounding Ajax, and quite a lot to get excited about, from both a technological and a business perspective. Technologically, Ajax gives expression to a lot of unrealized potential in the web browser technologies. Google and a few other major players are using Ajax to raise the expectations of the general public as to what a web application can do.

The classical "web application" that we have become used to is beginning to creak under the strain that increasingly sophisticated web-based services are placing on it. A variety of technologies are lining up to fill the gap with richer, smarter, or otherwise improved clients. Ajax is able to deliver this better, smarter richness using only technologies that are already installed on the majority of modern computers.

With Ajax, we are taking a bunch of dusty old technologies and stretching them well beyond their original scope. We need to be able to manage the complexity that we have introduced. This book will discuss the how-tos of the individual technologies but will also look at the bigger picture of managing large Ajax projects. We'll introduce Ajax *design patterns* throughout the book as well to help us get this job done. Design patterns help us to capture our knowledge and experience with a technology as we acquire it and to communicate it with others. By introducing regularity to a codebase, they can facilitate creating applications that are easy to modify and extend as requirements change. Design patterns are even a joy to work with!

1.1 Why Ajax rich clients?

Building a rich client interface is a bit more complicated than designing a web page. What is the incentive, then, for going this extra mile? What's the payoff? What is a rich client, anyway?

Two key features characterize a rich client: it's rich, and it's a client.

Let me explain a little more. *Rich* refers here to the interaction model of the client. A rich user interaction model is one that can support a variety of input methods and that responds intuitively and in a timely fashion. We could set a rather unambitious yardstick for this by saying that for user interaction to be rich, it must be as good as the current generation of desktop applications, such as word processors and spreadsheets. Let's take a look at what that would entail.

1.1.1 Comparing the user experiences

Take a few minutes to play with an application of your choice (other than a web browser), and count the types of user interaction that it offers. Come back here when you've finished. I'm going to discuss a spreadsheet as an example shortly, but the points I'll make are sufficiently generic that anything from a text editor up will do.

Finished? I am. While typing a few simple equations into my spreadsheet, I found that I could interact with it in a number of ways, editing data in situ, navigating the data with keyboard and mouse, and reorganizing data using drag and drop.

As I did these things, the program gave me feedback. The cursor changed shape, buttons lit up as I hovered over them, selected text changed color, highlighted windows and dialogs were represented differently, and so on (figure 1.1). That's what passes for rich interactivity these days. Arguably there's still some way to go, but it's a start.

So is the spreadsheet application a rich *client*? I would say that it isn't.

In a spreadsheet or similar desktop application, the logic and the data model are both executed in a closed environment, in which they can see each other very clearly but shut the rest of the world out (figure 1.2). My definition of a *client* is a program that communicates to a different, independent process, typically running on a server. Traditionally, the server is bigger, stronger, and better than the client, and it stores monstrously huge amounts of information. The client allows end users to view and modify this information, and if several clients are connected to the same server, it allows them to share that data. Figure 1.3 shows a simple schematic of a client/server architecture.

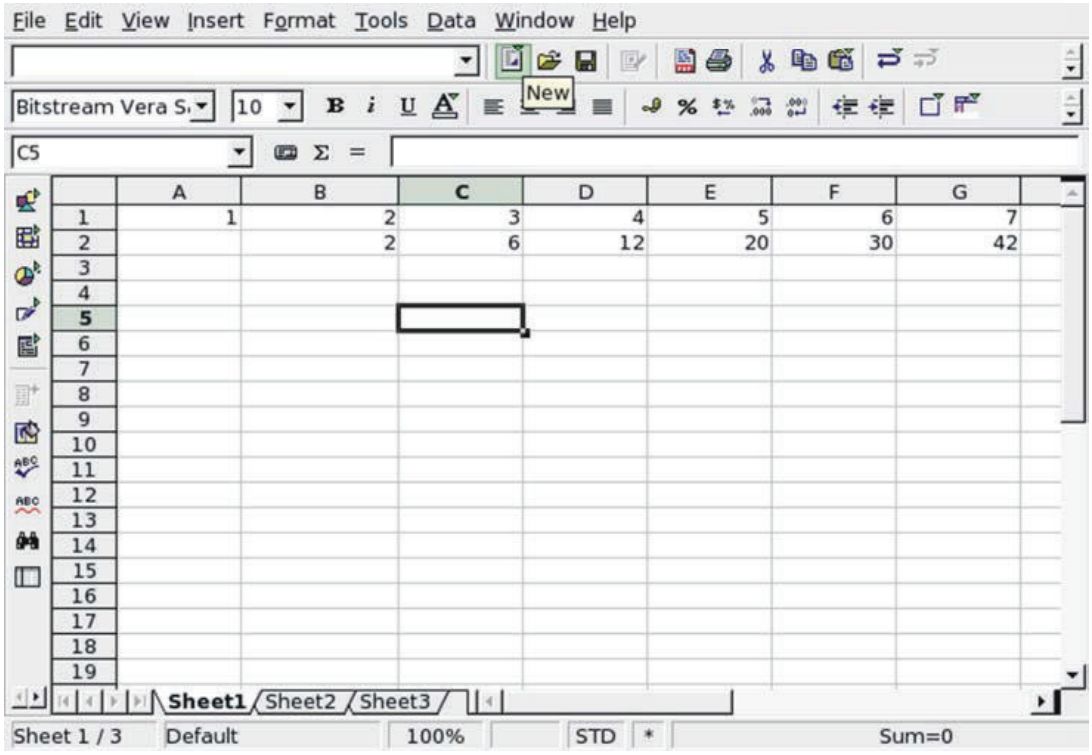

Figure 1.1 This desktop spreadsheet application illustrates a variety of possibilities for user interaction. The headers for the selected rows and columns are highlighted; buttons offer tooltips on mouseover; toolbars contain a variety of rich widget types; and the cells can be interactively inspected and edited.

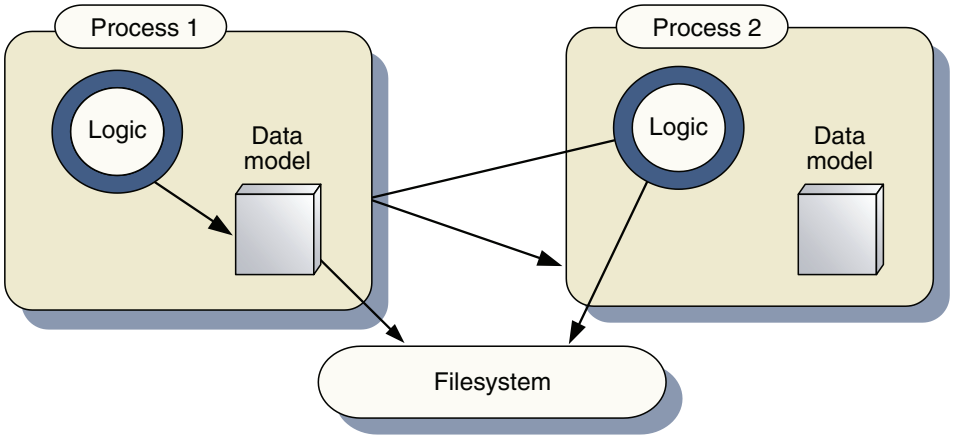

Figure 1.2 Schematic architectures for a standalone desktop application. The application runs in a process of its own, within which the data model and the program logic can "see" one another. A second running instance of the application on the same computer has no access to the data model of the first, except via the filesystem. Typically, the entire program state is stored in a single file, which is locked while the application is running, preventing any simultaneous exchange of information.

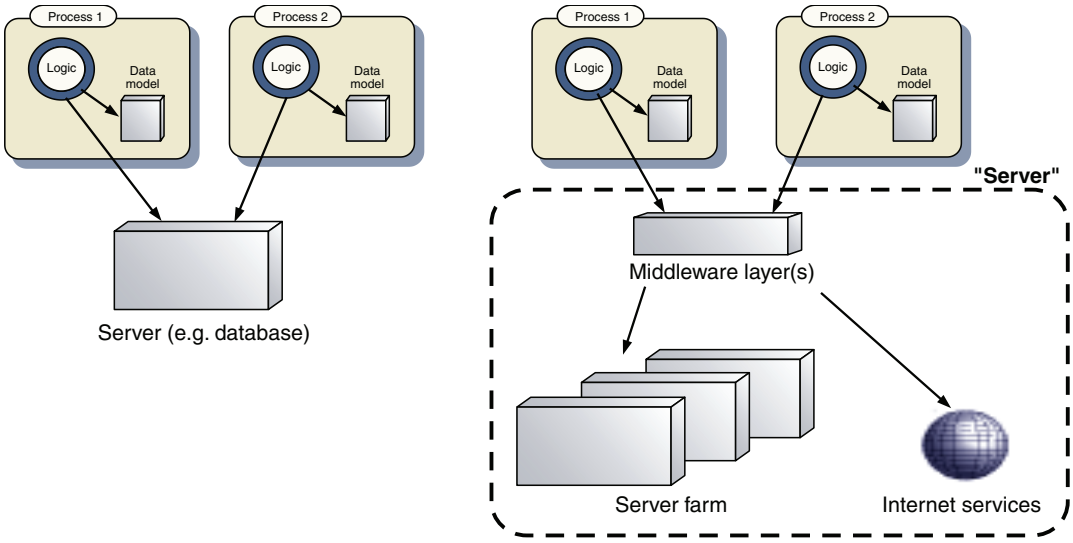

Figure 1.3 Schematic architectures for client/server systems and n-tier architectures. The server offers a shared data model, with which clients can interact. The clients still maintain their own partial data models, for rapid access, but these defer to the server model as the definitive representation of the business domain objects. Several clients can interact with the same server, with locking of resources handled at a fine-grain level of individual objects or database rows. The server may be a single process, as in the traditional client/server model of the early- to mid-1990s, or consist of several middleware tiers, external web services, and so on. In any case, from the client's perspective, the server has a single entry point and can be considered a black box.

In a modern n-tier architecture, of course, the server will communicate to further back-end servers such as databases, giving rise to middleware layers that act as both client and server. Our Ajax applications typically sit at the end of this chain, acting as client only, so we can treat the entire n-tier system as a single black box labeled "server" for the purposes of our current discussion.

My spreadsheet sits on its own little pile of data, stored locally in memory and on the local filesystem. If it is well architected, the coupling between data and presentation may be admirably loose, but I can't split it across the network or share it as such. And so, for our present purposes, it isn't a client.

Web browsers are clients, of course, contacting the web servers from which they request pages. The browser has some rich functionality for the purpose of managing the user's web browsing, such as back buttons, history lists, and tabs for storing several documents. But if we consider the web pages for a particular site as an application, then these generic browser controls are not related to the application any more than the Windows Start menu or window list are related to my spreadsheet.

Let's have a look at a modern web application. Simply because everyone has heard of it, we'll pick on Amazon, the bookseller (figure 1.4). I point my browser to the Amazon site, and, because it remembers who I am from my last visit, it shows me a friendly greeting, a list of recommended books, and information about my purchasing history.

Clicking on a title from the recommendations list leads me to a separate page (that is, the screen flickers and I lose sight of all the lists that I was viewing a few seconds earlier). This, too, is stuffed full of contextual information: reviews, second-hand prices for the book, links to similar authors, and titles of other books that I've recently checked out (figure 1.5).

In short, I'm presented with very rich, tightly interwoven information. And yet my only way of interacting with this information is through clicking hyperlinks and filling in text forms. If I fell asleep at the keyboard while browsing the site and awoke the next day, I wouldn't know that the new Harry Potter book had been released until I refreshed the entire page. I can't take my lists with me from one

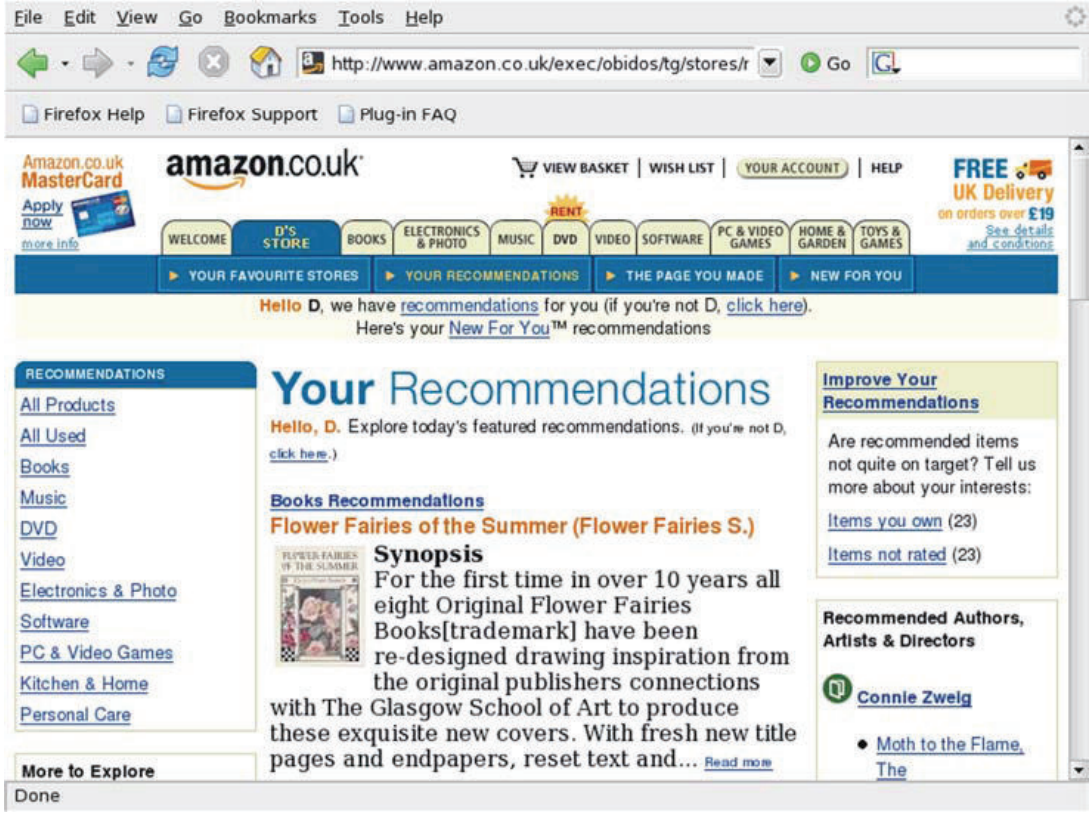

Figure 1.4 Amazon.com home page. The system has remembered who I am from a previous visit, and the navigational links are a mixture of generic boilerplate and personal information.

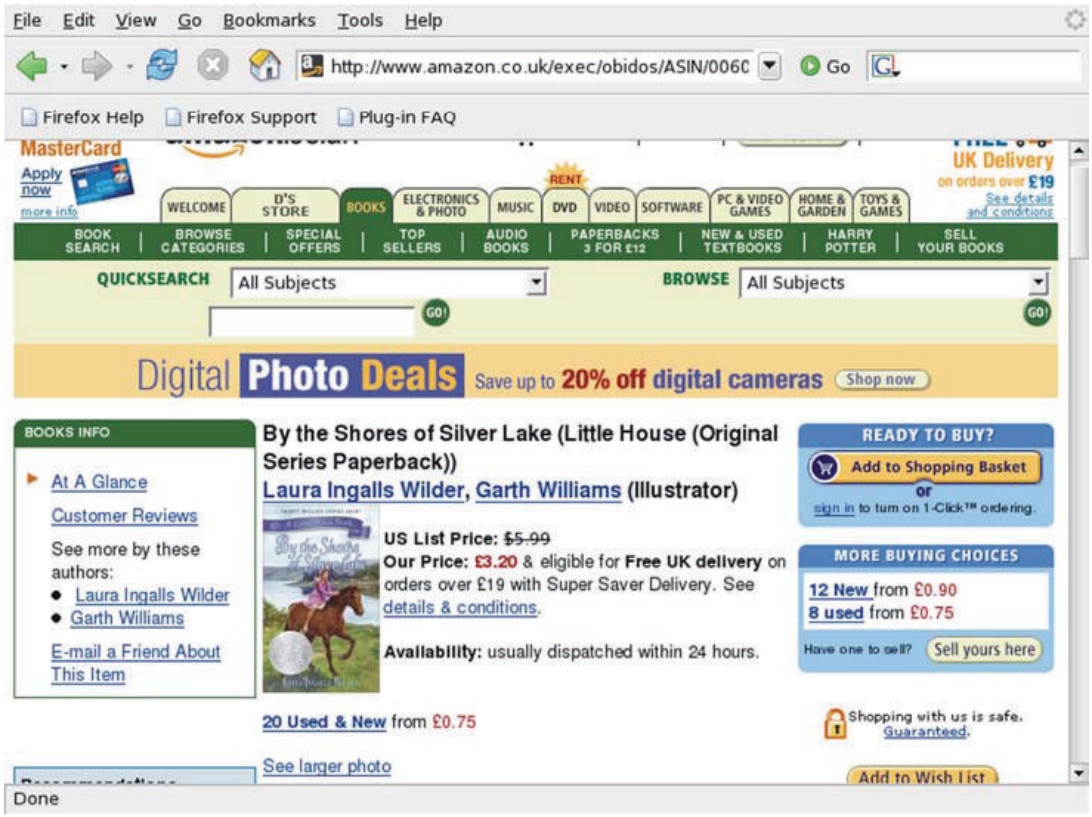

Figure 1.5 Amazon.com book details page. Again, a dense set of hyperlinks combines generic and personal information. Nonetheless, a significant amount of detail is identical to that shown in figure 1.4, which must, owing to the document-based operation of the web browser, be retransmitted with every page.

page to another, and I can't resize portions of the document to see several bits of content at once.

This is not to knock Amazon. It's doing a good job at working within some very tight bounds. But compared to the spreadsheet, the interaction model it relies on is unquestionably limiting.

So why are those limits present in modern web applications? There are sound technical reasons for the current situation, so let's take a look at them now.

1.1.2 Network latency

The grand vision of the Internet age is that all computers in the world interconnect as one very large computing resource. Remote and local procedure calls become indistinguishable, and issuers are no longer even aware of which physical

machine (or machines) they are working on, as they happily compute the folds in their proteins or decode extraterrestrial signals.

Remote and local procedure calls are not the same thing at all, unfortunately. Communications over a network are expensive (that is, they are slow and unreliable). When a non-networked piece of code is compiled or interpreted, the various methods and functions are coded as instructions stored in the same local memory as the data on which the methods operate (figure 1.6). Thus, passing data to a method and returning a result is pretty straightforward.

Under the hood, a lot of computation is going on at both ends of a network connection in order to send and receive data (figure 1.7). It's this computation that slows things down, more than the physical journey along the wire. The various stages of encoding and decoding cover aspects of the communication ranging from physical signals passing along the wire (or airwaves), translation of these signals as the 1s and 0s of binary data, error checking and re-sending, to the reassembling of the sequence, and ultimately the meaning, of the binary information.

The calling function's request must be encoded as an object, which is then serialized (that is, converted into a linear set of bytes). The serialized data is then passed to the application protocol (usually HTTP these days) and sent across the physical transport (a copper or fiber-optic cable, or a wireless connection of some sort).

On the remote machine, the application protocol is decoded, and the bytes of data deserialized, to create a copy of the request object. This object can then be applied to the data model and a response object generated. To communicate the response to the calling function, the serialization and transport layers must be navigated once more, eventually resulting in a response object being returned to the calling function.

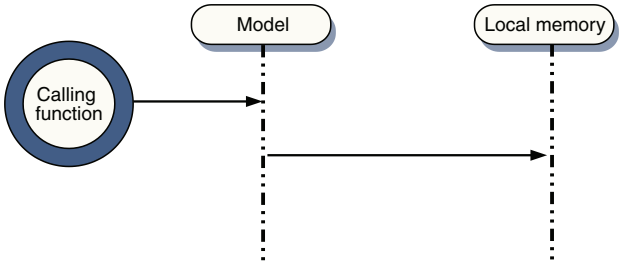

Figure 1.6 Sequence diagram of a local procedure call. Very few actors are involved here, as the program logic and the data model are both stored in local memory and can see each other directly.

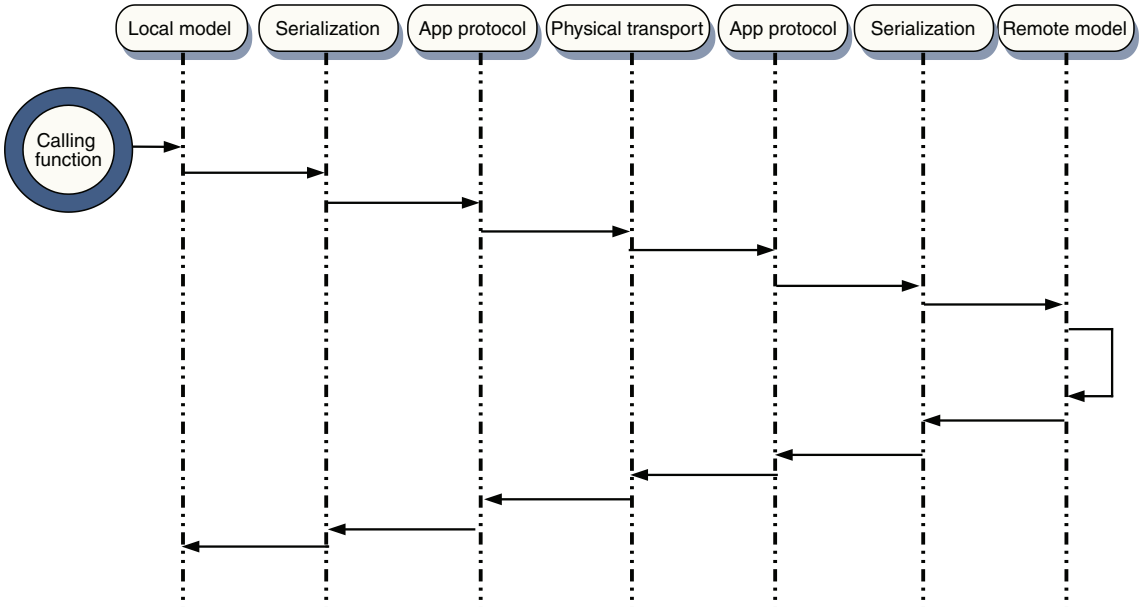

Figure 1.7 Sequence diagram of a remote procedure call. The program logic on one machine attempts to manipulate a data model on another machine.

These interactions are complex but amenable to automation. Modern programming environments such as Java and the Microsoft .NET Framework offer this functionality for free. Nonetheless, internally a lot of activity is going on when a remote procedure call (RPC) is made, and if such calls are made too freely, performance will suffer.

So, making a call over a network will never be as efficient as calling a local method in memory. Furthermore, the unreliability of the network (and hence the need to resend lost packets of information) makes this inefficiency variable and hard to predict. The responsiveness of the memory bus on your local machine is not only better but also very well defined in comparison.

But what does that have to do with usability? Quite a lot, as it turns out.

A successful computer UI does need to mimic our expectations of the real world *at the very basic level*. One of the most basic ground rules for interaction is that when we push, prod, or poke at something, it responds immediately. Slight delays between prodding something and the response can be disorienting and distracting, moving the user's attention from the task at hand to the UI itself.

Having to do all that extra work to traverse the network is often enough to slow down a system such that the delay becomes noticeable. In a desktop application, we need to make bad usability design decisions to make the application feel buggy or unresponsive, but in a networked application, we can get all that for free!

Because of the unpredictability of network latency, this perceived bugginess will come and go, and testing the responsiveness of the application can be harder, too. Hence, network latency is a common cause of poor interactivity in real-world applications.

1.1.3 Asynchronous interactions

There is only one sane response to the network latency problem available to the UI developer—assume the worst. In practical terms, we must try to make UI responses independent of network activity. Fortunately, a holding response is often sufficient, as long as it is timely. Let's take a trip to the physical world again. A key part of my morning routine is to wake my children up for school. I could stand over them prodding them until they are out of bed and dressed, but this is a time-consuming approach, leaving a long period of time in which I have very little to do (figure 1.8).

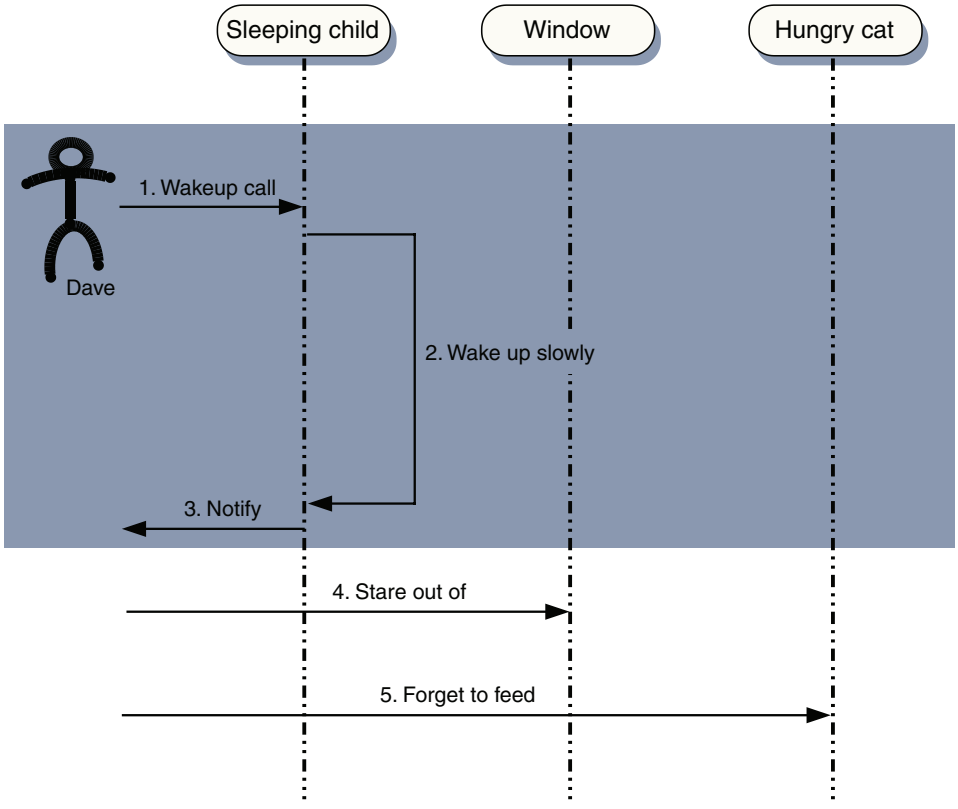

Figure 1.8 Sequence diagram of a synchronous response to user input, during my morning routine. In a sequence diagram, the passage of time is vertical. The height of the shaded area indicates the length of time for which I am blocked from further input.

I need to wake up my children, stare out the window, and ignore the cat. The children will notify me when they are properly awake by asking for breakfast. Like server-side processes, children are slow to wake. If I follow a synchronous interaction model, I will spend a long time waiting. As long as they are able to mutter a basic "Yes, I'm awake," I can happily move on to something else and check up on them later if need be.

In computer terms, what I'm doing here is spawning an asynchronous process, in a separate thread. Once they're started, my children will wake up by themselves in their own thread, and I, the parent thread, don't need to synchronize with them until they notify me (usually with a request to be fed). While they're waking up, I can't interact with them as if they were already up and dressed, but I can be confident that it will happen in due course (figure 1.9).

With any UI, it's a well-established practice to spawn an asynchronous thread to handle any lengthy piece of computation and let it run in the background while the user gets on with other things. The user is necessarily blocked while that thread is launched, but this can be done in an acceptably short span of time.

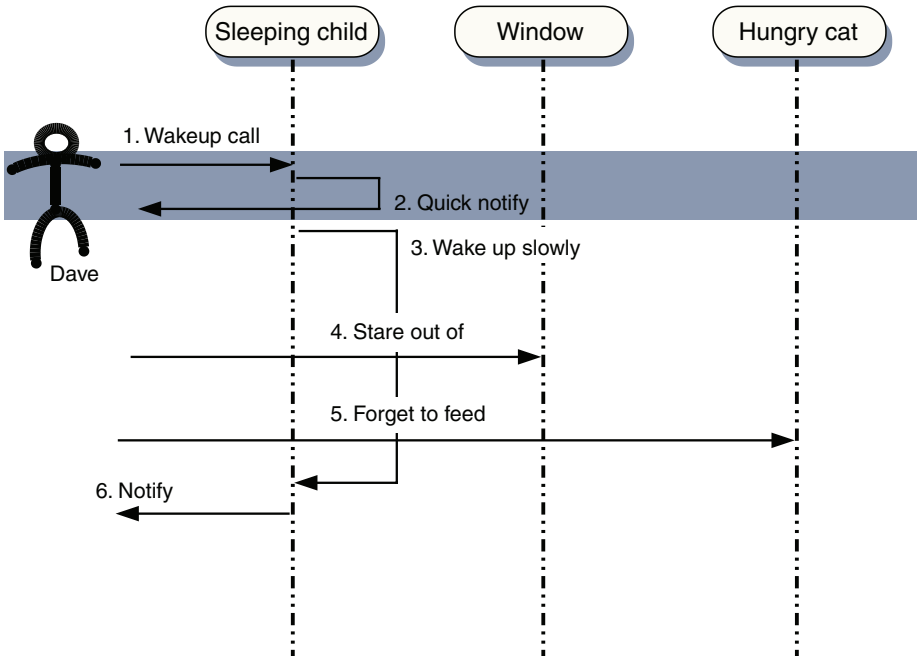

Figure 1.9 Sequence diagram of an asynchronous response to user input. If I follow an asynchronous input model, I can let the children notify me that they are starting to wake up. I can then continue with my other activities while the wakeup happens and remain blocked for a much shorter period of time.

Because of network latency, it is good practice to treat any RPC as potentially lengthy and handle it asynchronously.

This problem, and the solution, are both well established. Network latency was present in the old client/server model, causing poorly designed clients to freeze up inexplicably as they tried to reach an overloaded server. And now, in the Internet age, network latency causes your browser to "chug" frustratingly while moving between web pages. We can't get rid of latency, but we know how to deal with it—by processing the remote calls asynchronously, right?

Unfortunately for us web app developers, there's a catch. HTTP is a request-response protocol. That is, the client issues a request for a document, and the server responds, either by delivering the document, saying that it can't find it, offering an alternative location, or telling the client to use its cached copy, and so on. A request-response protocol is one-way. The client can make contact with the server, but the server cannot initiate a communication with the client. Indeed, the server doesn't remember the client from one request to the next.

The majority of web developers using modern languages such as Java, PHP, or .NET will be familiar with the concept of user sessions. These are an afterthought, bolted onto application servers to provide the missing server-side state in the HTTP protocol. HTTP does what it was originally designed for very well, and it has been adapted to reach far beyond that with considerable ingenuity. However, the key feature of our asynchronous callback solution is that the client gets notified twice: once when the thread is spawned and again when the thread is completed. Straightforward HTTP and the classic web application model can't do this for us.

The classic web app model, as used by Amazon, for example, is still built around the notion of pages. A document is displayed to the user, containing lists of links and/or form elements that allow them to drill down to further documents. Complex datasets can be interacted with in this way on a large scale, and as Amazon and others have demonstrated, the experience of doing so can be compelling enough to build a business on.

This model of interaction has become quite deeply ingrained in our way of thinking over the ten years or so of the commercial, everyday Internet. Friendly WYSIWYG web-authoring tools visualize our site as a collection of pages. Server-side web frameworks model the transition between pages as state transition diagrams. The classic web application is firmly wedded to the unavoidable lack of responsiveness when the page refreshes, without an easy recourse to the asynchronous handler solution.

But Amazon has built a successful business on top of its website. Surely the classic web application can't be that unusable? To understand why the web page works for Amazon but not for everyone, we ought to consider usage patterns.

1.1.4 Sovereign and transient usage patterns

It's futile to argue whether a bicycle is better than a sports utility vehicle. Each has its own advantages and disadvantages—comfort, speed, fuel consumption, vague psychological notions about what your mode of transport "says" about you as a person. When we look at particular use patterns, such as getting through the rush hour of a compact city center, taking a large family on vacation, or seeking shelter from the rain, we may arrive at a clear winner. The same is true for computer UIs.

Software usability expert Alan Cooper has written some useful words about usage patterns and defines two key usage modes: transient and sovereign. A *transient* application might be used every day, but only in short bursts and usually as a secondary activity. A *sovereign* application, in contrast, must cope with the user's full attention for several hours at a time.

Many applications are inherently transient or sovereign. A writer's word processor is a sovereign application, for example, around which a number of transient functions will revolve, such as the file manager (often embedded into the word processor as a file save or open dialog), a dictionary or spellchecker (again, often embedded), and an email or messenger program for communicating with colleagues. To a software developer, the text editor or Integrated Development Environment (IDE) is sovereign, as is the debugger.

Sovereign applications are also often used more intensely. Remember, a well-behaved UI should be invisible. A good yardstick for the intensity of work is the effect on the user's workflow of the UI stalling, thus reminding the user that it exists. If I'm simply moving files from one folder to another and hit a two-second delay, I can cope quite happily. If I encounter the same two-second delay while composing a visual masterpiece in a paint program, or in the middle of a heavy debugging session with some tricky code, I might get a bit upset.

Amazon is a transient application. So are eBay and Google—and most of the very large, public web-based applications out there. Since the dawn of the Internet, pundits have been predicting the demise of the traditional desktop office suite under the onslaught of web-based solutions. Ten years later, it hasn't happened. Web page–based solutions are good enough for transient use but not for sovereign use.

1.1.5 Unlearning the Web

Fortunately, modern web browsers resemble the original ideal of a client for remote document servers about as closely as a Swiss army knife resembles a neolithic flint hunting tool. Interactive gizmos, scripting languages, and plug-ins have been bolted on willy-nilly over the years in a race to create the most compelling browsing experience. (Have a look at www.webhistory.org/www.lists/www-talk.1993q1/0182.html to get a perspective on how far we've come. In 1993, a pre-Netscape Marc Andreessen tentatively suggested to Tim Berners-Lee and others that HTML might benefit from an image tag.)

A few intrepid souls have been looking at JavaScript as a serious programming language for several years, but on the whole, it is associated with faked-up alert dialogs and "click the monkey to win" banners.

Think of Ajax as a rehabilitation center for this misunderstood, ill-behaved child of the browser wars. By providing some guidance and a framework within which to operate, we can turn JavaScript into a helpful model citizen of the Internet, capable of enhancing the real usability of a web application—and without enraging the user or trashing the browser in the process. Mature, well-understood tools are available to help us do this. Design patterns are one such tool that we make frequent use of in our work and will refer to frequently in this book.

Introducing a new technology is a technical and social process. Once the technology is there, people need to figure out what to do with it, and a first step is often to use it as if it were something older and more familiar. Hence, early bicycles were referred to as "hobbyhorses" or "dandy horses" and were ridden by pushing one's feet along the ground. As the technology was exposed to a wider audience, a second wave of innovators would discover new ways of using the technology, adding improvements such as pedals, brakes, gears, and pneumatic tires. With each incremental improvement, the bicycle became less horse-like (figure 1.10).

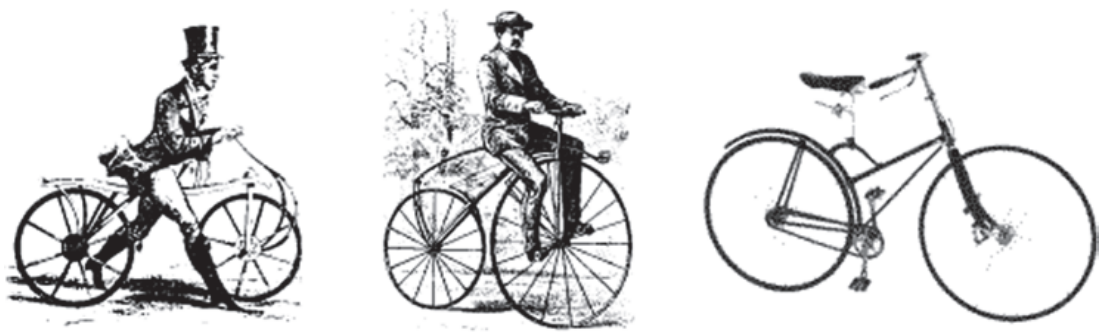

Figure 1.10 Development of the modern bicycle

The same processes are at work in web development today. The technologies behind Ajax have the ability to transform web pages into something radically new. Early attempts to use the Ajax technologies resembled the traditional web page document and have that neither-one-thing-nor-the-other flavor of the hobbyhorse. To grasp the potential of Ajax, we must let go of the concept of the web page and, in doing so, unlearn a lot of the assumptions that we have been making for the last few years. In the short few months since Ajax was christened, a lot of unlearning has been taking place.

1.2 The four defining principles of Ajax

The classic page-based application model is hard-wired into many of the frameworks that we use, and also into our ways of thinking. Let's take a few minutes to discover what these core assumptions are and how we need to rethink them to get the most out of Ajax.

1.2.1 The browser hosts an application, not content

In the classic page-based web application, the browser is effectively a dumb terminal. It doesn't know anything about where the user is in the greater workflow. All of that information is held on the web server, typically in the user's session. Server-side user sessions are commonplace these days. If you're working in Java or .NET, the server-side session is a part of the standard API, along with requests, responses, and Multipurpose Internet Mail Extensions (MIME) types. Figure 1.11 illustrates the typical lifecycle of a classic web application.

When the user logs in or otherwise initializes a session, several server-side objects are created, representing, say, the shopping basket and the customer credentials if this is an e-commerce site. At the same time, the home page is dished up to the browser, in a stream of HTML markup that mixes together standard boilerplate presentation and user-specific data and content such as a list of recently viewed items.

Every time the user interacts with the site, another document is sent to the browser, containing the same mixture of boilerplate and data. The browser dutifully throws the old document away and displays the new one, because it is dumb and doesn't know what else to do.

When the user hits the logout link or closes the browser, the application exits and the session is destroyed. Any information that the user needs to see the next time she or he logs on will have been handed to the persistence tier by

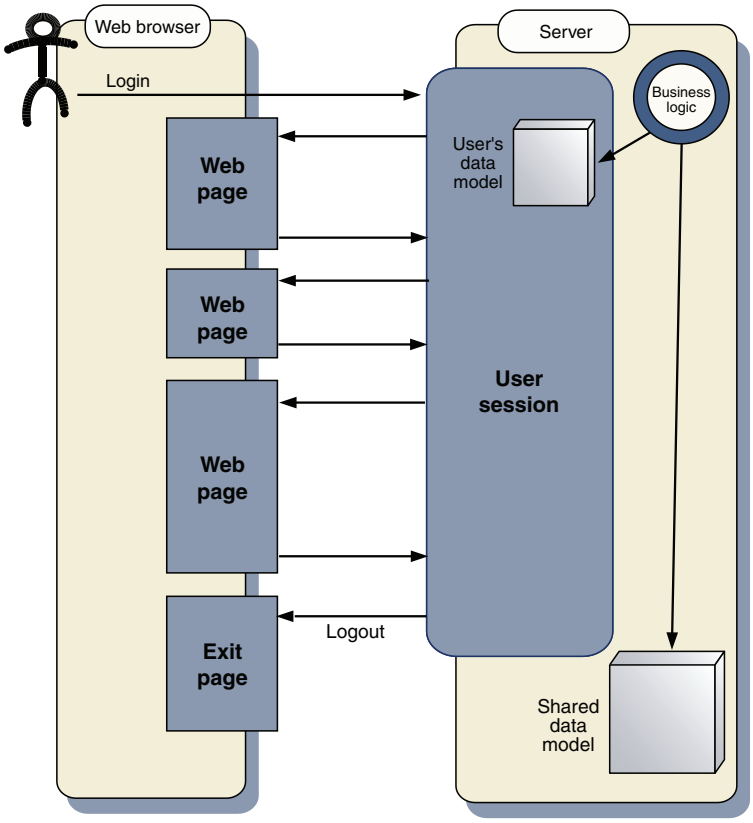

Figure 1.11 Lifecycle of a classic web application. All the state of the user's "conversation" with the application is held on the web server. The user sees a succession of pages, none of which can advance the broader conversation without going back to the server.

now. An Ajax application moves some of the application logic to the browser, as figure 1.12 illustrates.

When the user logs in, a more complex document is delivered to the browser, a large proportion of which is JavaScript code. This document will stay with the user throughout the session, although it will probably alter its appearance considerably while the user is interacting with it. It knows how to respond to user input and is able to decide whether to handle the user input itself or to pass a request on to the web server (which has access to the system database and other resources), or to do a combination of both.

Because the document persists over the entire user session, it can store state. A shopping basket's contents may be stored in the browser, for example, rather than in the server session.

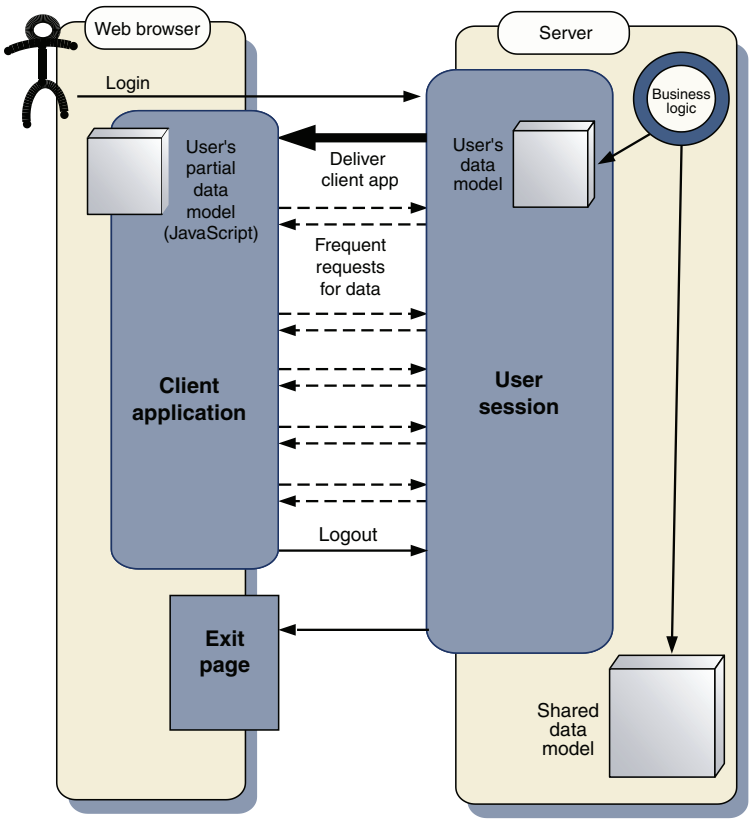

Figure 1.12 Lifecycle of an Ajax application. When the user logs in, a client application is delivered to the browser. This application can field many user interactions independently, or else send requests to the server behind the scenes, without interrupting the user's workflow.

1.2.2 The server delivers data, not content

As we noted, the classic web app serves up the same mixture of boilerplate, content, and data at every step. When our user adds an item to a shopping basket, all that we really need to respond with is the updated price of the basket or whether anything went wrong. As illustrated in figure 1.13, this will be a very small part of the overall document.

An Ajax-based shopping cart could behave somewhat smarter than that, by sending out asynchronous requests to the server. The boilerplate, the navigation lists, and other features of the page layout are all there already, so the server needs to send back only the relevant data.

The Ajax application might do this in a number of ways, such as returning a fragment of JavaScript, a stream of plain text, or a small XML document. We'll

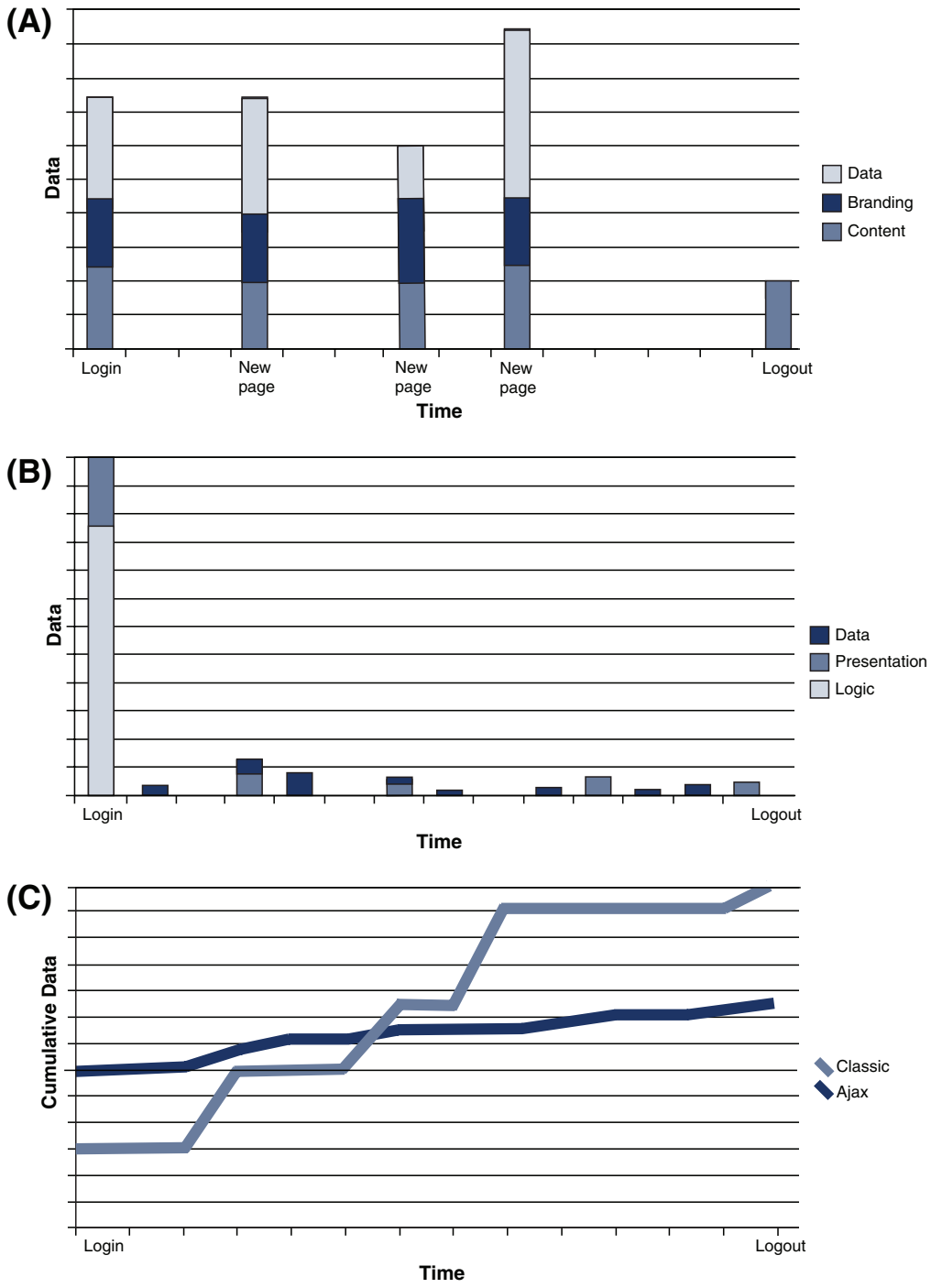

Figure 1.13 Breakdown of the content delivered (A) to a classic web application and (B) to an Ajax application. As the application continues to be used, cumulative traffic (C) increases.

look at the pros and cons of each in detail in chapter 5. Suffice it to say for now that any one of these formats will be much smaller than the mish-mash returned by the classic web application.

In an Ajax application, the traffic is heavily front-loaded, with a large and complex client being delivered in a single burst when the user logs in. Subsequent communications with the server are far more efficient, however. For a transient application, the cumulative traffic may be less for a conventional web page application, but as the average length of interaction time increases, the bandwidth cost of the Ajax application becomes less than that of its classic counterpart.

1.2.3 User interaction with the application can be fluid and continuous

A web browser provides two input mechanisms out of the box: hyperlinks and HTML forms.

Hyperlinks can be constructed on the server and preloaded with Common Gateway Interface (CGI) parameters pointed at dynamic server pages or servlets. They can be dressed up with images and Cascading Style Sheets (CSS) to provide rudimentary feedback when the mouse hovers over them. Given a good web designer, hyperlinks can be made to look like quite fancy UI components.

Form controls offer a basic subset of the standard desktop UI components: input textboxes, checkboxes and radio buttons, and drop-down lists. Several likely candidates are missing, though. There are no out-of-the-box tree controls, editable grids, or combo-boxes provided. Forms, like hyperlinks, point at server-side URLs.

Alternatively, hyperlinks and form controls can be pointed at JavaScript functions. It's a common technique in web pages to provide rudimentary form validation in JavaScript, checking for empty fields, out-of-range numbers, and so on, before submitting data to the server. These JavaScript functions persist only as long as the page itself and are replaced when the page submits.

While the page is submitting, the user is effectively in limbo. The old page may still be visible for a while, and the browser may even allow the user to click on any visible links, but doing so will produce unpredictable results and may wreak havoc with the server-side session. The user is generally expected to wait until the page is refreshed, often with a set of choices similar to those that were snatched away from them seconds earlier. After all, adding a pair of trousers to the shopping basket is unlikely to modify the top-level categories from "menswear," "women's wear," "children's," and "accessories."

Let's take the shopping cart example again. Because our Ajax shopping cart sends data asynchronously, users can drop things into it as fast as they can click. If the cart's client-side code is robust, it will handle this load easily, and the users can get on with what they're doing.

There is no cart to drop things into, of course, just an object in session on the server. Users don't want to know about session objects while shopping, and the cart metaphor provides a more comfortable real-world description of what's taking place. Switching contexts between the metaphor and direct access to the computer is distracting to users. Waiting for a page to refresh will jerk them back to the reality of sitting at a computer for a short time (figure 1.14), and our Ajax implementation avoids doing this. Shopping is a transient activity, but if we consider a different business domain, for example, a high-pressure help desk scenario or a complex engineering task, then the cost of disrupting the workflow every few seconds with a page refresh is prohibitive.

The second advantage of Ajax is that we can hook events to a wider range of user actions. More sophisticated UI concepts such as drag-and-drop become feasible, bringing the UI experience fully up to par with the desktop application widget sets. From a usability perspective, this freedom is important not so much because it allows us to exercise our imagination, but because it allows us to blend the user interaction and server-side requests more fully.

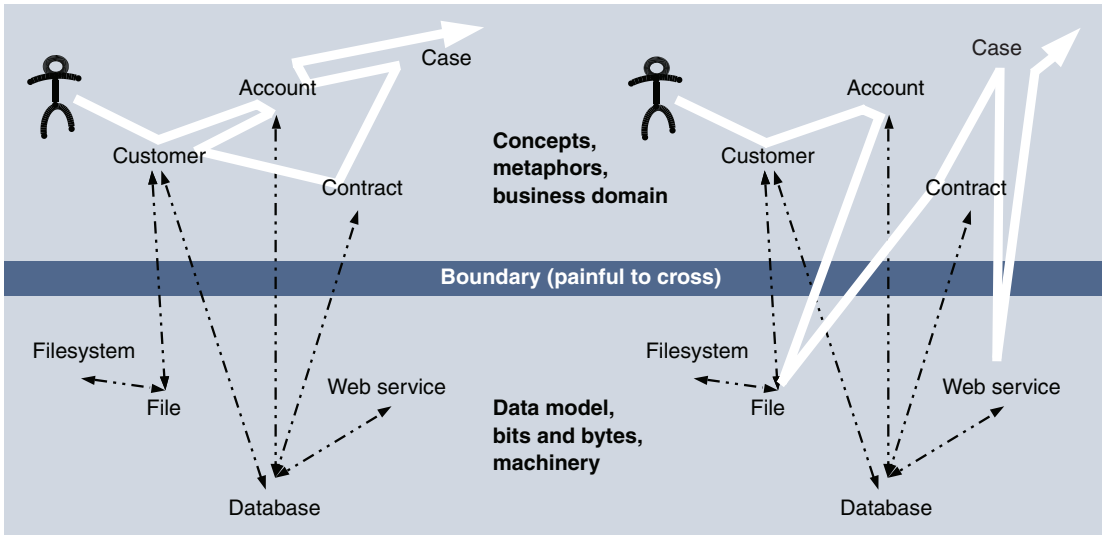

Figure 1.14 Interrupting the user's workflow to process events. The user deals with two types of object: those relating to their business, and those relating to the computer system. Where the user is forced to switch between the two frequently, disorientation and lack of productivity may occur.

To contact the server in a classic web application, we need to click a hyperlink or submit a form, and then wait. This interrupts the user's workflow. In contrast, contacting the server in response to a mouse movement or drag, or a keystroke, allows the server to work alongside the user. Google Suggest (www.google.com/webhp?complete=1) is a very simple but effective example of this: responding to users keystrokes as they type into the search box and contacting the server to retrieve and display a list of likely completions for the phrases, based on searches made by other users of the search engine worldwide. We provide a simple implementation of a similar service in chapter 8.

1.2.4 This is real coding and requires discipline

Classic web applications have been making use of JavaScript for some time now, to add bells and whistles around the edge of their pages. The page-based model prevents any of these enhancements from staying around for too long, which limits the uses to which they can be put. This catch-22 situation has led, unfairly, to JavaScript getting a reputation as a trivial, hacky sort of language, looked down upon by the serious developers.

Coding an Ajax application is a different matter entirely. The code that you deliver when users launch the application must run until they close it, without breaking, without slowing down, and without generating memory leaks. If we're aiming at the sovereign application market, then this means several hours of heavy usage. To meet this goal, we must write high-performance, maintainable code, using the same discipline and understanding that is successfully applied to the server tiers.

The codebase will also typically be larger than anything written for a classic web application. Good practices in structuring the codebase become important. The code may become the responsibility of a team rather than an individual, bringing up issues of maintainability, separation of concerns, and common coding styles and patterns.

An Ajax application, then, is a complex functional piece of code that communicates efficiently with the server while the user gets on with work. It is clearly a descendent of the classic page-based application, but the similarity is no stronger than that between the early hobbyhorse and a modern touring bike. Bearing these differences in mind will help you to create truly compelling web applications.

1.3 Ajax rich clients in the real world

So much for the theory. Ajax is already being used to create real applications, and the benefit of the Ajax approach can already be seen. It's still very much early days—the bicycles of a few far-sighted individuals have pedals and solid rubber tires, and some are starting to build disc brakes and gearboxes, so to speak. The following section surveys the current state of the art and then looks in detail at one of the prominent early adopters to see where the payoff in using Ajax lies.

1.3.1 Surveying the field

Google has done more than any other company to raise the profile of Ajax applications (and it, like the majority of adopters, was doing so before the name Ajax was coined). Its GMail service was launched in beta form in early 2004. Along with the extremely generous mailbox size, the main buzz around GMail was the UI, which allowed users to open several mail messages at once and which updated mailbox lists automatically, even while the user was typing in a message. Compared with the average web mail system offered by most Internet service providers (ISPs) at the time, this was a major step forward. Compared with the corporate mail server web interfaces of the likes of Microsoft Outlook and Lotus Notes, GMail offered most of the functionality without resorting to heavy, troublesome ActiveX controls or Java applets, making it available across most platforms and locations, rather than the corporate user's carefully preinstalled machine.

Google has followed this up with further interactive features, such as Google Suggest, which searches the server for likely completions for your query as you type, and Google Maps, an interactive zoomable map used to perform location-based searches. At the same time, other companies have begun to experiment with the technology, such as Flickr's online photo-sharing system, now part of Yahoo!

The applications we have discussed so far are testing the water. They are still transient applications, designed for occasional use. There are signs of an emerging market for sovereign Ajax applications, most notably the proliferation of frameworks in recent months. We look at a few of these in detail in chapter 3, and attempt to summarize the current state of the field in appendix C.

There are, then, sufficient signals to suggest that Ajax is taking hold of the market in a significant way. We developers will play with any new technology for its own sake, but businesses like Google and Yahoo! will join in only if there are compelling business reasons. We've already outlined many of the theoretical advantages of Ajax. In the following section, we'll take apart Google Maps, in order to see how the theory stacks up.

1.3.2 Google Maps

Google Maps is a cross between a map viewer and a search engine. Initially, the map shows the entire United States (figure 1.15). The map can be queried using free text, allowing drill-down to specific street addresses or types of amenity such as hotels and restaurants (figure 1.16).

The search feature functions as a classic web app, refreshing the entire page, but the map itself is powered by Ajax. Clicking on individual links from a hotel search will cause additional pop-ups to be displayed on the fly, possibly even scrolling the map slightly to accommodate them. The scrolling of the map itself is

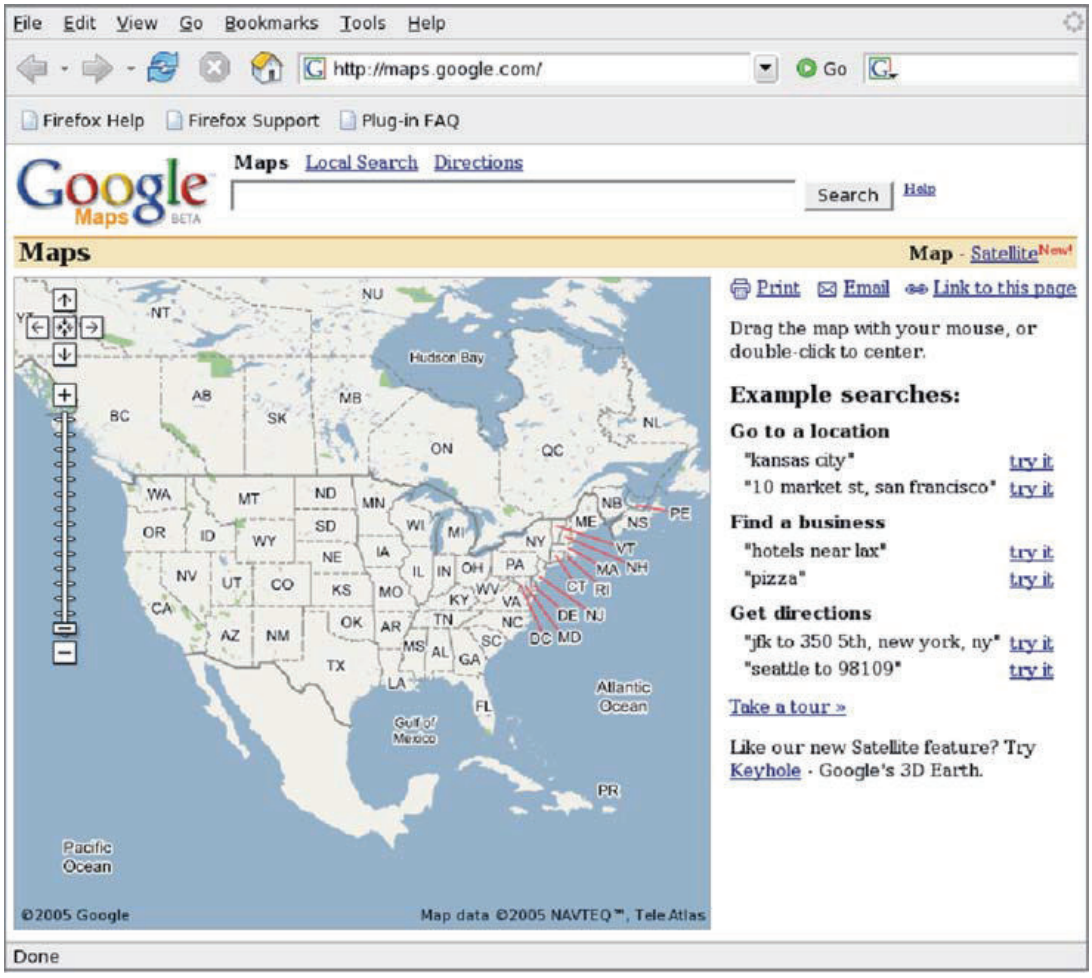

Figure 1.15 The Google Maps home page offers a scrolling window on a zoomable map of the United States, alongside the familiar Google search bar. Note that the zoom control is positioned on top of the map rather than next to it, allowing the user to zoom without taking his eyes off the map.

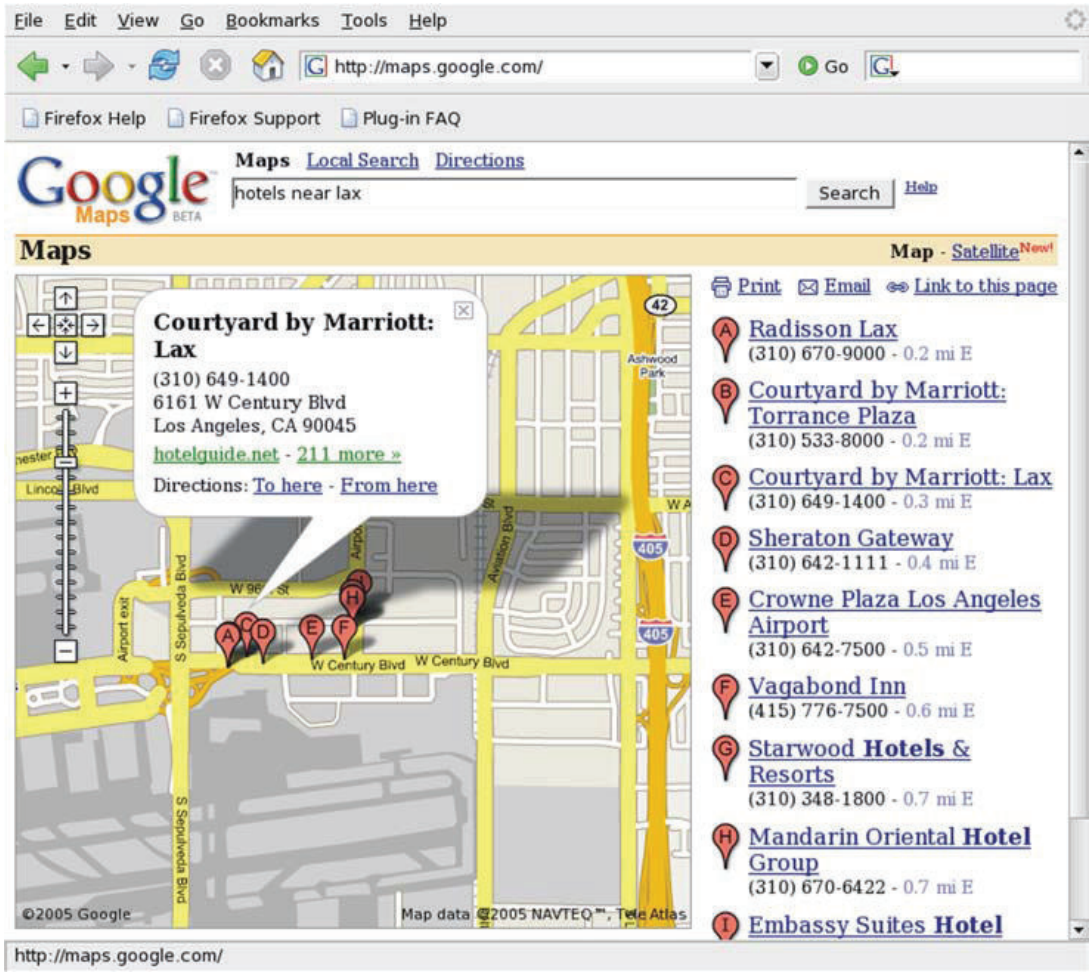

Figure 1.16 Google Maps hotel search. Note the traditional use of the DHTML technologies to create shadows and rich tooltip balloons. Adding Ajax requests makes these far more dynamic and useful.

the most interesting feature of Google Maps. The user can drag the entire map by using the mouse. The map itself is composed of small tiled images, and if the user scrolls the map far enough to expose a new tile, it will be asynchronously downloaded. There is a noticeable lag at times, with a blank white area showing initially, which is filled in once the map tile is loaded; however, the user can continue to scroll, triggering fresh tile requests, while the download takes place. The map tiles are cached by the browser for the extent of a user's session, making it much quicker to return to a part of the map already visited.

Looking back to our discussions of usability, two important things are apparent. First, the action that triggers the download of new map data is not a specific

click on a link saying "fetch more maps" but something that the user is doing anyway, namely, moving the map around. The user workflow is uninterrupted by the need to communicate with the server. Second, the requests themselves are asynchronous, meaning that the contextual links, zoom control, and other page features remain accessible while the map is gathering new data.

Internet-based mapping services are nothing new. If we looked at a typical pre-Ajax Internet mapping site, we would see a different set of interaction patterns. The map would typically be divided into tiles. A zoom control, and perhaps sideways navigation links at the map's edges, might be provided. Clicking on any of these would invoke a full-screen refresh, resulting in a similar page hosting different map tiles. The user workflow would be interrupted more, and after looking at Google Maps, the user would find the site slow and frustrating.

Turning to the server-side, both services are undoubtedly backed by some powerful mapping solutions. Both serve up map tiles as images. The conventional web server of the pre-Ajax site is continually refreshing boilerplate code when the user scrolls, whereas Google Maps, once up and running, serves only the required data, in this case image tiles that aren't already cached. (Yes, the browser will cache the images anyway, providing the URL is the same, but browser caching still results in server traffic when checking for up-to-date data and provides a less-reliable approach than programmatic caching in memory.) For a site with the prominent exposure of Google, the bandwidth savings must be considerable.

To online services such as Google, ease of use is a key feature in getting users to visit their service and to come back again. And the number of page impressions is a crucial part of the bottom line for the business. By introducing a better UI with the flexibility that Ajax offers, Google has clearly given traditional mapping services something to worry about. Certainly other factors, such as the quality of the back-end service, come into play, but other things being equal, Ajax can offer a strong business advantage.

We can expect the trend for this to rise as public exposure to richer interfaces becomes more prevalent. As a marketable technology, Ajax looks to have a bright future for the next few years. However, other rich client technologies are looking to move into this space, too. Although they are largely outside the scope of this book, it's important that we take a look at them before concluding our overview.

1.4 Alternatives to Ajax

Ajax meets a need in the marketplace for richer, more responsive web-based clients that don't need any local installation. It isn't the only player in that space, though, and in some cases, it isn't even the most appropriate choice. In the following section, we'll briefly describe the main alternatives.

1.4.1 Macromedia Flash-based solutions

Macromedia's Flash is a system for playing interactive movies using a compressed vector graphics format. Flash movies can be streamed, that is, played as they are downloaded, allowing users to see the first bits of the movie before the last bits have arrived. Flash movies are interactive and are programmed with ActionScript, a close cousin of JavaScript. Some support for input form widgets is also provided, and Flash can be used for anything from interactive games to complex business UIs. Flash has very good vector graphics support, something entirely absent from the basic Ajax technology stack.

Flash has been around for ages and is accessed by a plug-in. As a general rule, relying on a web browser plug-in is a bad idea, but Flash is *the* web browser plug-in, with the majority of browsers bundling it as a part of the installation. It is available across Windows, Mac OS X, and Linux, although the installation base on Linux is probably smaller than for the other two platforms.

For the purposes of creating rich clients with Flash, two very interesting technologies are Macromedia's Flex and the open source Laszlo suite, both of which provide simplified server-side frameworks for generating Flash-based business UIs. Both frameworks use Java/Java 2 Enterprise Edition (J2EE) on the server side. For lower-level control over creating Flash movies dynamically, several toolkits, such as PHP's libswf module, provide core functionality.

1.4.2 Java Web Start and related technologies

Java Web Start is a specification for bundling Java-based web applications on a web server in such a way that a desktop process can find, download, and run them. These applications can be added as hyperlinks, allowing seamless access from a Web Start–savvy web browser. Web Start is bundled with the more recent Java runtimes, and the installation process will automatically enable Web Start on Internet Explorer and Mozilla-based browsers.

Once downloaded, Web Start applications are stored in a managed "sandbox" in the filesystem and automatically updated if a new version is made available. This allows them to be run while disconnected from the network and reduces

network traffic on reload, making the deployment of heavy applications weighing several megabytes a possibility. Applications are digitally signed, and the user may choose to grant them full access to the filesystem, network ports, and other resources.

Traditionally, Web Start UIs are written in the Java Swing widget toolkit, about which strong opinions are held on both sides. The Standard Widget Toolkit (SWT) widgets used to power IBM's Eclipse platform can also be deployed via Web Start, although this requires a bit more work.

Microsoft's .NET platform offers a similar feature called No Touch Deployment, promising a similar mix of easy deployment, rich UIs, and security.

The main downside to both technologies is the need to have a runtime preinstalled. Of course, any rich client needs a runtime, but Flash and Ajax (which uses the web browser itself as a runtime) use runtimes that are commonly deployed. Java and .NET runtimes are both very limited in their distribution at present and can't be relied on for a public web service.

1.5 Summary

We've discussed the differences between transient and sovereign applications and the requirements of each. Transient applications need to deliver the goods, but, when users are using them, they have already stepped out of their regular flow of work, and so a certain amount of clunkiness is acceptable. Sovereign applications, in contrast, are designed for long-term intensive use, and a good interface for a sovereign application must support the users invisibly, without breaking their concentration on the task at hand.

The client/server and related n-tier architectures are essential for collaborative or centrally coordinated applications, but they raise the specter of network latency, with its ability to break the spell of user productivity. Although a general-purpose solution to the conflict between the two exists in asynchronous remote event handling, the traditional request-response model of the classic web application is ill suited to benefit from it.

We've set a goal for ourselves, and for Ajax, in this chapter of delivering usable sovereign applications through a web browser, thereby satisfying the goals of user productivity, networking, and effortless, centralized maintenance of an application all at once. In order for this mission to succeed, we need to start thinking about our web pages and applications in a fundamentally different way. We've identified the key ideas that we need to learn and those that we need to unlearn:

- The browser hosts an application, not content.
- The server delivers data, not content.
- The user interacts continuously with the application, and most requests to the server are implicit rather than explicit.
- Our codebase is large, complex, and well structured. It is a first-class citizen in our architecture, and we must take good care of it.

The next chapter will unpack the key Ajax technologies and get our hands dirty with some code. The rest of the book will look at important design principles that can help us to realize these goals.

1.6 Resources

To check out some of our references in greater depth, here are URLs to several of the articles that we've referred to in this chapter:

- Jesse James Garrett christened Ajax on February 18, 2005, in this article: www.adaptivepath.com/publications/essays/archives/000385.php
- Alan Cooper's explanation of sovereign and transient applications can be found here: www.cooper.com/articles/art_your_programs_posture.htm
- Google Maps can be found here if you live in the United States: http://maps.google.com

  and here if you live in the United Kingdom: http://maps.google.co.uk

  and here if you live on the moon: http://moon.google.com

The images of the bicycle were taken from the Pedaling History website: www.pedalinghistory.com

2 First steps with Ajax

This chapter covers

- Introducing the technologies behind Ajax
- Using Cascading Style Sheets to define look and feel
- Using the Document Object Model to define the user interface structure
- Using XMLHttpRequest to asynchronously contact the server
- Putting the pieces together

In chapter 1 we focused on users and how Ajax can assist them in their daily activities. Most of us are developers, and so, having convinced ourselves that Ajax is a Good Thing, we need to know how to work with it. The good news is that, as with many brand-new, shiny technologies, most of this process will be reasonably familiar already, particularly if you've worked with the Internet.

In this chapter, we'll explain the Ajax technology. We'll discuss the four technological cornerstones of Ajax and how they relate to one another, using code examples to demonstrate how each technology works and how everything fits together.

You might like to think of this chapter as the "hello world" section of the book, in which we introduce the core technologies using some simple examples. We're more interested here in just getting things to work; we'll start to look at the bigger picture in chapter 3. If you're already familiar with some or all of the Ajax technologies, you may want to skim these sections. If you're new to Ajax and to web client programming, these introductions should be sufficient to orient you for the rest of the book.

2.1 The key elements of Ajax

Ajax isn't a single technology. Rather, it's a collection of four technologies that complement one another. Table 2.1 summarizes these technologies and the role that each has to play.

Table 2.1 The key elements of Ajax

JavaScript	JavaScript is a general-purpose scripting language designed to be embedded inside applications. The JavaScript interpreter in a web browser allows programmatic interaction with many of the browser's inbuilt capabilities. Ajax applications are written in JavaScript.
Cascading Style Sheets (CSS)	CSS offers a way of defining reusable visual styles for web page elements. It offers a simple and powerful way of defining and applying visual styling consistently. In an Ajax application, the styling of a user interface may be modified interactively through CSS.
Document Object Model (DOM)	The DOM presents the structure of web pages as a set of programmable objects that can be manipulated with JavaScript. Scripting the DOM allows an Ajax application to modify the user interface on the fly, effectively redrawing parts of the page.
XMLHttpRequest object	The (misnamed) XMLHttpRequest object allows web programmers to retrieve data from the web server as a background activity. The data format is typically XML, but it works well with any text-based data. While XMLHttpRequest is the most flexible general-purpose tool for this job, there are other ways of retrieving data from the server, too, and we'll cover them all in this chapter.

We saw in chapter 1 how an Ajax application delivers a complex, functioning application up front to users, with which they then interact. JavaScript is the glue that is used to hold this application together, defining the user workflow and business logic of the application. The user interface is manipulated and refreshed by using JavaScript to manipulate the Document Object Model (DOM), continually redrawing and reorganizing the data presented to the users and processing their mouse- and keyboard-based interactions. Cascading Style Sheets (CSS) provide a consistent look and feel to the application and a powerful shorthand for the programmatic DOM manipulation. The XMLHttpRequest object (or a range of similar mechanisms) is used to talk to the server asynchronously, committing user requests and fetching up-to-date data while the user works. Figure 2.1 shows how the technologies fit together in Ajax.

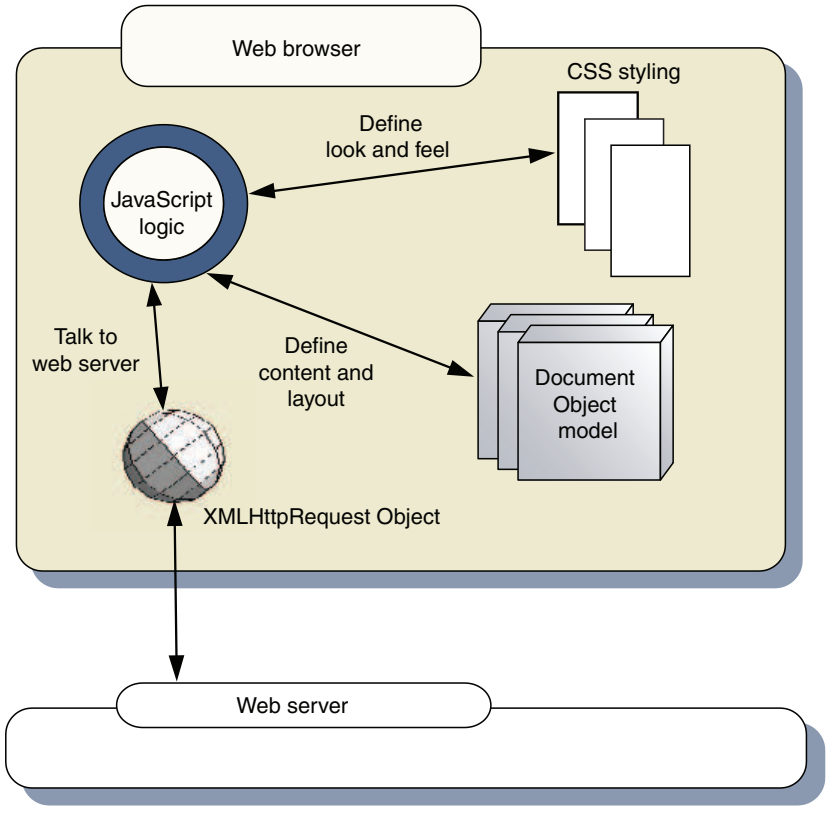

Figure 2.1 The four main components of Ajax: JavaScript defines business rules and program flow. The Document Object Model and Cascading Style Sheets allow the application to reorganize its appearance in response to data fetched in the background from the server by the XMLHttpRequest object or its close cousins.

Three of the four technologies—CSS, DOM, and JavaScript—have been collectively referred to as Dynamic HTML, or DHTML for short. DHTML was the Next Big Thing around 1997, but not surprisingly in this industry, it never quite lived up to its initial promise. DHTML offered the ability to create funky, interactive interfaces for web pages, yet it never overcame the issue of the full-page refresh. Without going back to talk to the server, there was only so much that we could do. Ajax makes considerable use of DHTML, but by adding the asynchronous request, it can extend the longevity of a web page considerably. By going back to the server while the interface is doing its stuff, without interruption, Ajax makes a great difference to the end result.

Rather conveniently, all of these technologies are already preinstalled in most modern web browsers, including Microsoft's Internet Explorer; the Mozilla/Gecko family of browsers, including Firefox, Mozilla Suite, Netscape Navigator, and Camino; the Opera browser; Apple's Safari; and its close cousin Konqueror, from the UNIX KDE desktop. Inconveniently, the implementations of these technologies are frustratingly different in some of the fine details and will vary from version to version, but this situation has been improving over the last five years, and we have ways of coping cleanly with cross-browser incompatibilities.

Every modern operating system comes with a modern browser preinstalled. So the vast majority of desktop and laptop computers on the planet are already primed to run Ajax applications, a situation that most Java or .NET developers can only dream about. (The browsers present in PDAs and Smartphones generally offer a greatly cut-down feature list and won't support the full range of Ajax technologies, but differences in screen size and input methods would probably be an issue even if they did. For now, Ajax is principally a technology for desktop and laptop machines.)

We'll begin by reviewing these technologies in isolation and then look at how they interoperate. If you're a seasoned web developer, you'll probably know a lot of this already, in which case you might like to skip ahead to chapter 3, where we begin to look at managing the technologies by using design patterns.

Let's start off our investigations by looking at JavaScript.

2.2 Orchestrating the user experience with JavaScript

The central player in the Ajax toolkit is undoubtedly JavaScript. An Ajax application downloads a complete client into memory, combining data and presentation and program logic, and JavaScript is the tool used to implement that logic.

JavaScript is a general-purpose programming language of mixed descent, with a superficial similarity to the C family of languages.

JavaScript can be briefly characterized as a loosely typed, interpreted, general-purpose scripting language. *Loosely typed* means that variables are not declared specifically as strings, integers, or objects, and the same variable may be assigned values of different types. For example, the following is valid code:

```
var x=3.1415926;
x='pi';
```

The variable `x` is defined first as a numeric value and reassigned a string value later.

Interpreted means that it is not compiled into executable code, but the source code is executed directly. When deploying a JavaScript application, you place the source code on the web server, and the source code is transmitted directly across the Internet to the web browser. It's even possible to evaluate snippets of code on the fly:

```
var x=eval('7*5');
```

Here we have defined our calculation as a piece of text, rather than two numbers and an arithmetic operator. Calling `eval()` on this text interprets the JavaScript it contains, and returns the value of the expression. In most cases, this simply slows the program execution down, but at times the extra flexibility that it brings can be useful.

General purpose means that the language is suitable for use with most algorithms and programming tasks. The core JavaScript language contains support for numbers, strings, dates and times, arrays, regular expressions for text processing, and mathematical functions such as trigonometry and random number generation. It is possible to define structured objects using JavaScript, bringing design principles and order to more complex code.

Within the web browser environment, parts of the browser's native functionality, including CSS, the DOM, and the XMLHttpRequest objects, are exposed to the JavaScript engine, allowing page authors to programmatically control the page to a greater or lesser degree. Although the JavaScript environment that we encounter in the browser is heavily populated with browser-specific objects, the underlying language is just that, a programming language.

This isn't the time or place for a detailed tutorial on JavaScript basics. In appendix B we take a closer look at the language and outline the fundamental differences between JavaScript and the C family of languages, including its

namesake, Java. JavaScript examples are sprinkled liberally throughout this book, and several other books already exist that cover the language basics (see our Resources section at the end of this chapter).

Within the Ajax technology stack, JavaScript is the glue that binds all the other components together. Having a basic familiarity with JavaScript is a prerequisite for writing Ajax applications. Being fluent in JavaScript and understanding its strengths will allow you to take full advantage of Ajax.

We'll move on now to Cascading Style Sheets, which control the visual style of elements on a web page.

2.3 Defining look and feel using CSS

Cascading Style Sheets are a well-established part of web design, and they find frequent use in classic web applications as well as in Ajax. A stylesheet offers a centralized way of defining categories of visual styles, which can then be applied to individual elements on a page very concisely. In addition to the obvious styling elements such as color, borders, background images, transparency, and size, stylesheets can define the way that elements are laid out relative to one another and simple user interactivity, allowing quite powerful visual effects to be achieved through stylesheets alone.

In a classic web application, stylesheets provide a useful way of defining a style in a single place that can be reused across many web pages. With Ajax, we don't think in terms of a rapid succession of pages anymore, but stylesheets still provide a helpful repository of predefined looks that can be applied to elements dynamically with a minimum of code. We'll work through a few basic CSS examples in this section, but first, let's look at how CSS rules are defined.

CSS styles a document by defining rules, usually in a separate file that is referred to by the web page being styled. Style rules can also be defined inside a web page, but this is generally considered bad practice.

A style rule consists of two parts: the *selector* and the *style declaration*. The selector specifies which elements are going to be styled, and the style declaration declares which style properties are going to be applied. Let's say that we want to make all our level-1 headings in a document (that is, the `<H1>` tags) appear red. We can declare a CSS rule to do this:

```
h1 { color: red }
```

The selector here is very simple, applying to all `<H1>` tags in the document. The style declaration is also very simple, modifying a single style property. In practice,

both the selector and the style declaration can be considerably more complex. Let's look at the variations in each, starting with the selector.

2.3.1 CSS selectors

In addition to defining a type of HTML tag to apply a style to, we can limit the rule to those within a specific context. There are several ways of specifying the context: by HTML tag type, by a declared class type, or by an element's unique ID.

Let's look at tag-type selectors first. For example, to apply the above rule only to `<H1>` tags that are contained within a `<DIV>` tag, we would modify our rule like this:

```
div h1 { color: red; }
```

These are also referred to as element-based selectors, because they decide whether or not a DOM element is styled based on its element type. We can also define classes for styling that have nothing to do with the HTML tag type. For example, if we define a style class called `callout`, which is to appear in a colored box, we could write

```
.callout { border: solid blue 1px; background-color: cyan }
```

To assign a style class to an element, we simply declare a class attribute in the HTML tag, such as

```
<div>I'll appear as a normal bit of text</div>
<div class='callout'>And I'll appear as a callout!</div>
```

Elements can be assigned more than one class. Suppose that we define an additional style class `loud` as

```
.loud { color: orange }
```

and apply both the styles in a document like so:

```
<div class='loud'>I'll be bright orange</div>
<div class='callout'>I'll appear as a callout</div>
<div class='callout loud'>
And I'll appear as an unappealing mixture of both!
</div>
```

The third `<div>` element will appear with orange text in a cyan box with a blue border. It is also possible to combine CSS styles to create a pleasing and harmonious design!

We can combine classes with element-based rules, to define a class that operates only on particular tag types. For example:

```
span.highlight { background-color: yellow }
```

will be applied only to tags with a declared class attribute of highlight. Other tags, or other types of tag with class='highlight', will be unaffected.

We can also use these in conjunction with the parent-child selectors to create very specific rules:

```
div.prose span.highlight { background-color: yellow }
```

This rule will be applied only to tags of class highlight that are nested within tags of class prose.

We can specify rules that apply only to an element with a given unique ID, as specified by the id attribute in the HTML. No more than one element in an HTML document should have a given ID assigned to it, so these selectors are typically used to select a single element on a page. To draw attention to a close button on a page, for example, we might define a style:

```
#close { color: red }
```

CSS also allows us to define styles based on pseudo-selectors. A web browser defines a limited number of pseudo-selectors. We'll present a few of the more useful ones here. For example:

```
*:first-letter {
  font-size: 500%;
  color: red;
  float: left;
}
```

will draw the first letter of any element in a large bold red font. We can tighten up this rule a little, like this:

```
 p.illuminated:first-letter {
  font-size: 500%;
  color: red;
  float: left;
}
```

The red border effect will now apply only to <p> elements with a declared class of illuminated. Other useful pseudo-selectors include first-line, and hover, which modifies the appearance of hyperlinks when the mouse pointer passes over them. For example, to make a link appear in yellow when under the mouse pointer, we could write the following rule:

```
a:hover{ color:yellow; }
```

That covers the bases for CSS selectors. We've already introduced several style declarations informally in these examples. Let's have a closer look at them now.

2.3.2 CSS style properties

Every element in an HTML page can be styled in a number of ways. The most generic elements, such as the `<DIV>` tag, can have dozens of stylings applied to them. Let's look briefly at a few of these.

The text of an element can be styled in terms of the color, the font size, the heaviness of the font, and the typeface to use. Multiple options can be specified for fonts, to allow graceful degradation in situations where a desired font is not installed on a client machine. To style a paragraph in gray, terminal-style text, we could define a styling:

```
.robotic{
  font-size: 14pt;
  font-family: courier new, courier, monospace;
  font-weight: bold;
  color: gray;
}
```

Or, more concisely, we could amalgamate the font elements:

```
.robotic{
  font: bold 14pt courier new, courier, monospace;
  color: gray;
}
```

In either case, the multiple styling properties are written in a key-value pair notation, separated by semicolons.

CSS can define the layout and size (often referred to as the *box-model)* of an element, by specifying margins and padding elements, either for all four sides or for each side individually:

```
.padded{ padding: 4px; }
.eccentricPadded {
  padding-bottom: 8px;
  padding-top: 2px;
  padding-left: 2px;
  padding-right: 16px;
  margin: 1px;
}
```

The dimensions of an element can be specified by the `width` and `height` properties. The position of an element can be specified as either absolute or relative. Absolutely positioned elements can be positioned on the page by setting the `top` and `left` properties, whereas relatively positioned elements will flow with the rest of the page.

Background colors can be set to elements using the `background-color` property. In addition, a background image can be set, using the `background-image` property:

```
.titlebar{ background-image: url(images/topbar.png); }
```

Elements can be hidden from view by setting either `visibility:hidden` or `display:none`. In the former case, the item will still occupy space on the page, if relatively positioned, whereas in the latter case, it won't.

This covers the basic styling properties required to construct user interfaces for Ajax applications using CSS. In the following section, we'll look at an example of putting CSS into practice.

2.3.3 A simple CSS example

We've raced through the core concepts of Cascading Style Sheets. Let's try putting them into practice now. CSS can be used to create elegant graphic design, but in an Ajax application, we're often more concerned with creating user interfaces that mimic desktop widgets. As a simple example of this type of CSS use, figure 2.2 shows a folder widget styled using CSS.

CSS performs two roles in creating the widget that we see on the right in figure 2.2. Let's look at each of them in turn.

Using CSS for layout

The first job is the positioning of the elements. The outermost element, representing the window as a whole, is assigned an absolute position:

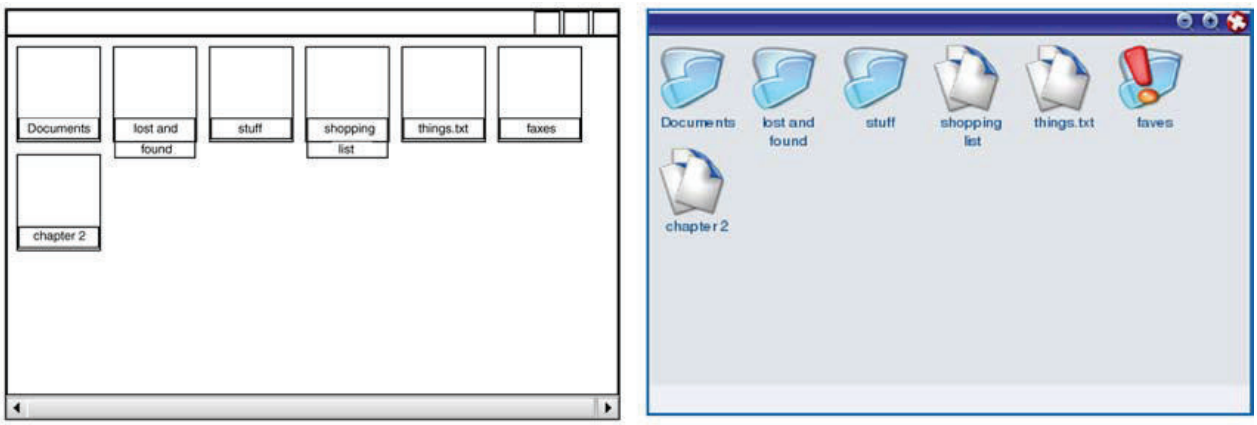

Figure 2.2 Using CSS to style a user interface widget. Both screenshots were generated from identical HTML, with only the stylesheets altered. The stylesheet used on the left retains only the positioning elements, whereas the stylesheet used to render the right adds in the decorative elements, such as colors and images.

```
div.window{
  position: absolute;
  overflow: auto;
  margin: 8px;
  padding: 0px;
  width: 420px;
  height: 280px;
}
```

Within the content area, the icons are styled using the `float` property so as to flow within the confines of their parent element, wrapping around to a new line where necessary:

```
div.item{
  position: relative;
  height: 64px;
  width: 56px;
  float: left;
  padding: 0px;
  margin: 8px;
}
```

The `itemName` element, which is nested inside the `item` element, has the text positioned below the icon by setting an upper margin as large as the icon graphic:

```
div.item div.itemName{
  margin-top: 48px;
  font: 10px verdana, arial, helvetica;
  text-align: center;
}
```

Using CSS for styling

The second job performed by CSS is the visual styling of the elements. The graphics used by the items in the folder are assigned by class name, for example:

```
div.folder{
  background:
    transparent url(images/folder.png)
    top left no-repeat;
}
div.file{
  background:
    transparent url(images/file.png)
    top left no-repeat;
}
div.special{
  background:
    transparent url(images/folder_important.png)
    top left no-repeat;
}
```

The `background` property of the icon styles is set to not repeat itself and be positioned at the top left of the element, with transparency enabled. (Figure 2.2 is rendered using Firefox. Transparency of .png images under Internet Explorer is buggy, with a number of imperfect proposed workarounds available. The forthcoming Internet Explorer 7 fixes these bugs, apparently. If you need cross-browser transparent images, we suggest the use of .gif images at present.)

Individual items declare two style classes: The generic item defines their layout in the container, and a second, more specific one defines the icon to be used. For example:

```
<div class='item folder'>
<div class='itemName'>stuff</div>
</div>
<div class='item file'>
<div class='itemName'>shopping list</div>
</div>
```

All the images in the styling are applied as background images using CSS. The titlebar is styled using an image as tall as the bar and only 1 pixel wide, repeating itself horizontally:

```
div.titlebar{
  background-color: #0066aa;
  background-image: url(images/titlebar_bg.png);
  background-repeat: repeat-x;
  ...
}
```

The full HTML for this widget is presented in listing 2.1.

Listing 2.1 window.html

```
<html>
<head>
<link rel='stylesheet' type='text/css'
  href='window.css' />                          Link to stylesheet
</head>
<body>
<div class='window'>   <-- Top-level window element
  <div class='titlebar'>
    <span class='titleButton' id='close'></span>
    <span class='titleButton' id='max'></span>      Titlebar buttons
    <span class='titleButton' id='min'></span>
  </div>
  <div class='contents'>
    <div class='item folder'>
      <div class='itemName'>Documents</div>
    </div>
    <div class='item folder'>
```

```
      <div class='itemName'>lost and found</div>
    </div>
    <div class='item folder'>
      <div class='itemName'>stuff</div>
    </div>
    <div class='item file'>
      <div class='itemName'>shopping list</div>
    </div>
    <div class='item file'>
      <div class='itemName'>things.txt</div>
    </div>
    <div class='item special'>
      <div class='itemName'>faves</div>
    </div>
    <div class='item file'>
      <div class='itemName'>chapter 2</div>
    </div>
  </div>
</div>
</body>
</html>
```

An icon inside a window

The HTML markup defines the structure of the document, not the look. It also defines points in the document through which the look can be applied, such as class names, unique IDs, and even the tag types themselves. Reading the HTML, we can see how each element relates to the other in terms of containment but not the eventual visual style. Editing the stylesheet can change the look of this document considerably while retaining the structure, as figure 2.2 has demonstrated. The complete stylesheet for the widget is shown in listing 2.2.

Listing 2.2 window.css

```
div.window{
  position: absolute;
  overflow: auto;
  background-color: #eeefff;
  border: solid #0066aa 2px;
  margin: 8px;
  padding: 0px;
  width: 420px;
  height: 280px;
}
div.titlebar{
  background-color: #0066aa;
  background-image:
  url(images/titlebar_bg.png);
  background-repeat: repeat-x;
```

❶ **Geometry of element**

❷ **Background texture**

```
  color:white;
  border-bottom: solid black 1px;
  width: 100%;
  height: 16px;
  overflow:hidden;
}
span.titleButton{
  position: relative;
  height: 16px;
  width: 16px;
  padding: 0px;
  margin: 0px 1px; 0px 1px;
  float:right;
}
span.titleButton#min{
  background: transparent
    url(images/min.png) top left no-repeat;
}
span.titleButton#max{
  background: transparent
    url(images/max.png) top left no-repeat;
}
span.titleButton#close{
  background: transparent
    url(images/close.png) top left no-repeat;
}
div.contents {
  background-color: #e0e4e8;
  overflow: auto;
  padding: 2px;
  height:240px;
}
div.item{
  position : relative;
  height : 64px;
  width: 56px;
  float: left;
  color : #004488;
  font-size: 18;
  padding: 0px;
  margin: 4px;
}
div.item div.itemName {
  margin-top: 48px;
  font: 10px verdana, arial, helvetica;
  text-align: center;
}
div.folder{
  background: transparent
    url(images/folder.png) top left no-repeat;
}
```

❸ **Flow layout**

❹ **Text placement**

```
div.file{
  background: transparent
    url(images/file.png) top left no-repeat;
}
div.special{
  background: transparent
    url(images/folder_important.png)
    top left no-repeat;
}
```

We've already looked at a number of the tricks that we've employed in this stylesheet to tune the look and feel of individual elements. We've highlighted a few more here, to demonstrate the breadth of concerns to which CSS can be applied: on-screen placement ❶, texturing elements ❷, assisting in layout of elements ❸, and placing text relative to accompanying graphics ❹.

CSS is an important part of the web developer's basic toolkit. As we've demonstrated here, it can be applied just as easily to the types of interfaces that an Ajax application requires as to the more design-oriented approach of a static brochure-style site.

2.4 Organizing the view using the DOM

The Document Object Model (DOM) exposes a document (a web page) to the JavaScript engine. Using the DOM, the document structure, as seen in figure 2.3, can be manipulated programmatically. This is a particularly useful ability to have at our disposal when writing an Ajax application. In a classic web application, we are regularly refreshing the entire page with new streams of HTML from the server, and we can redefine the interface largely through serving up new HTML. In an Ajax application, the majority of changes to the user interface will be made using the DOM. HTML tags in a web page are organized in a tree structure. The root of the tree is the <HTML> tag, which represents the document. Within this, the <BODY> tag, which represents the document body, is the root of the visible document structure. Inside the body, we find table, paragraph, list, and other tag types, possibly with other tags inside them.

A DOM representation of a web page is also structured as a tree, composed of elements or nodes, which may contain child nodes within them, and so on recursively. The JavaScript engine exposes the root node of the current web page through the global variable *document*, which serves as the starting point for all our DOM manipulations. The DOM element is well defined by the W3C specification.

It has a single parent element, zero or more child elements, and any number of attributes, which are stored as an associative array (that is, by a textual key such as width or style rather than a numerical index). Figure 2.3 illustrates the abstract structure of the document shown in listing 2.2, as seen using the Mozilla DOM Inspector tool (see appendix A for more details).

The relationship between the elements in the DOM can be seen to mirror that of the HTML listing. The relationship is two-way. Modifying the DOM will alter the HTML markup and hence the presentation of the page.

This provides a top-level view of what the DOM looks like. In the following section, we'll see how the DOM is exposed to the JavaScript interpreter and how to work with it.

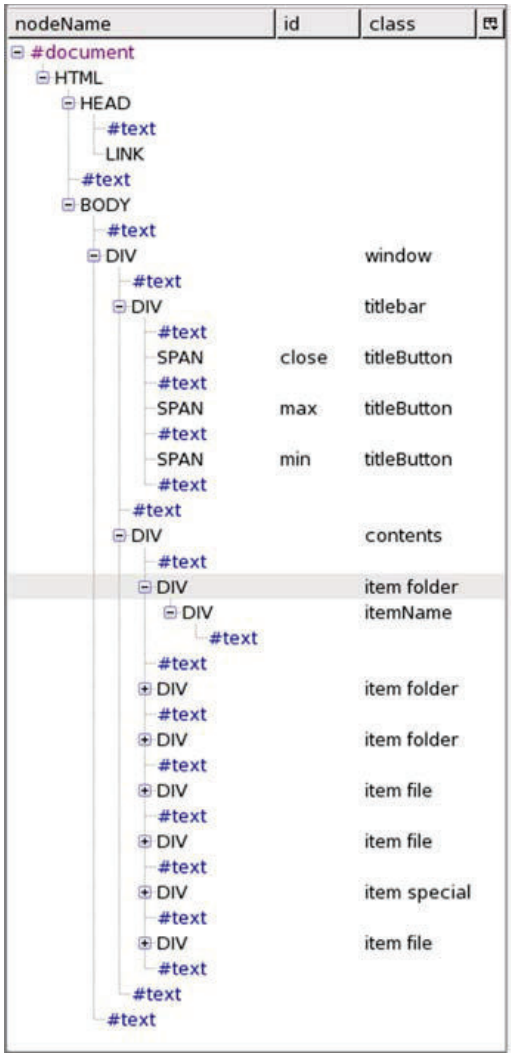

Figure 2.3
The DOM presents an HTML document as a tree structure, with each element representing a tag in the HTML markup.

2.4.1 Working with the DOM using JavaScript

In any application, we want to modify the user interface as users work, to provide feedback on their actions and progress. This could range from altering the label or color of a single element, through popping up a temporary dialog, to replacing large parts of the application screen with an entirely new set of widgets. By far the most usual is to construct a DOM tree by feeding the browser with declarative HTML (in other words, writing an HTML web page).

The document that we showed in listing 2.2 and figure 2.3 is rather large and complex. Let's start our DOM manipulating career with a small step. Suppose that we want to show a friendly greeting to the user. When the page first loads, we don't know his name, so we want to be able to modify the structure of the page to add his name in later, possibly to manipulate the DOM nodes programmatically. Listing 2.3 shows the initial HTML markup of this simple page.

Listing 2.3 Ajax "hello" page

```
<html>
<head>
<link rel='stylesheet' type='text/css'
  href='hello.css' />          <-- ❶ Link to stylesheet
<script type='text/javascript'
  src='hello.js'></script>     ❷ Link to JavaScript
</head>
<body>
<p id='hello'>hello</p>
<div id='empty'></div>         ❸ Empty element
</body>
```

We have added references to two external resources: a Cascading Style Sheet ❶ and a file containing some JavaScript code ❷. We have also declared an empty `<div>` element with an ID ❸, into which we can programmatically add further elements.

Let's look at the resources that we've linked to. The stylesheet defines some simple stylings for differentiating between different categories of item in our list by modifying the font and color (listing 2.4).

Listing 2.4 hello.css

```
.declared{
  color: red;
  font-family: arial;
  font-weight: normal;
```

```
  font-size: 16px;
}
.programmed{
  color: blue;
  font-family: helvetica;
  font-weight: bold;
  font-size: 10px;
}
```

We define two styles, which describe the origin of our DOM nodes. (The names of the styles are arbitrary. We called them that to keep the example easy to understand, but we could have just as easily called them `fred` and `jim`.) Neither of these style classes is used in the HTML, but we will apply them to elements programmatically. Listing 2.5 shows the JavaScript to accompany the web page in listing 2.4. When the document is loaded, we will programmatically style an existing node and create some more DOM elements programmatically.

Listing 2.5 hello.js

```
window.onload=function(){
  var hello=document.getElementById('hello');    <-- Find element by ID
  hello.className='declared';

  var empty=document.getElementById('empty');
  addNode(empty,"reader of");
  addNode(empty,"Ajax in Action!");

  var children=empty.childNodes;
  for (var i=0;i<children.length;i++){
    children[i].className='programmed';
  }

  empty.style.border='solid green 2px';          Style node
  empty.style.width="200px";                     directly
}
function addNode(el,text){
  var childEl=document.createElement("div");     <-- Create new element
  el.appendChild(childEl);
  var txtNode=document.createTextNode(text);     <-- Create text element
  childEl.appendChild(txtNode);
}
```

The JavaScript code is a bit more involved than the HTML or the stylesheet. The entry point for the code is the `window.onload()` function, which will be called programmatically once the entire page has been loaded. At this point, the DOM tree

has been built, and we can begin to work with it. Listing 2.5 makes use of several DOM manipulation methods, to alter attributes of the DOM nodes, show and hide nodes, and even create completely new nodes on the fly. We won't cover every DOM manipulation method here—have a look at our resources section for that—but we'll walk through some of the more useful ones in the next few sections.

2.4.2 Finding a DOM node

The first thing that we need to do in order to work on a DOM with JavaScript is to find the elements that we want to change. As mentioned earlier, all that we are given to start with is a reference to the root node, in the global variable `document`. Every node in the DOM is a child, (or grandchild, great-grandchild, and so on) of `document`, but crawling down the tree, step by step, could be an arduous process in a big complicated document. Fortunately, there are some shortcuts. The most commonly used of these is to tag an element with a unique ID. In the `onload()` function in listing 2.5 we want to find two elements: the paragraph element, in order to style it, and the empty `<div>` tag, in order to add contents to it. Knowing, this, we attached unique ID attributes to each in the HTML, thus:

```
<p id='hello'>
```

and

```
<div id='empty'></div>
```

Any DOM node can have an ID assigned to it, and the ID can then be used to get a programmatic reference to that node in one function call, wherever it is in the document:

```
var hello=document.getElementById('hello');
```

Note that this is a method of a Document object. In a simple case like this (and even in many complicated cases), you can reference the current Document object as `document`. If you end up using IFrames, which we'll discuss shortly, then you have multiple Document objects to keep track of, and you'll need to be certain which one you're querying.

In some situations, we do want to walk the DOM tree step by step. Since the DOM nodes are arranged in a tree structure, every DOM node will have no more than one parent but any number of children. These can be accessed by the `parentNode` and `childNodes` properties. `parentNode` returns another DOM node object, whereas `childNodes` returns a JavaScript array of nodes that can be iterated over; thus:

```
var children=empty.childNodes;
for (var i=0;i<children.length;i++){
  ...
}
```

A third method worth mentioning allows us to take a shortcut through documents that we haven't tagged with unique IDs. DOM nodes can also be searched for based on their HTML tag type, using `getElementsByTagName()`. For example, `document.getElementsByTagName("UL")` will return an array of all `<UL>` tags in the document.

These methods are useful for working with documents over which we have relatively little control. As a general rule, it is safer to use `getElementById()` than `getElementsByTagName()`, as it makes fewer assumptions about the structure and ordering of the document, which may change independently of the code.

2.4.3 Creating a DOM node

In addition to reorganizing existing DOM nodes, there are cases where we want to create completely new nodes and add them to the document (say, if we're creating a message box on the fly). The JavaScript implementations of the DOM give us methods for doing that, too.

Let's look at our example code (listing 2.5) again. The DOM node with ID `'empty'` does indeed start off empty. When the page loads, we created some content for it dynamically. Our `addNode()` function uses the standard `document.createElement()` and `document.createTextNode()` methods. `createElement()` can be used to create any HTML element, taking the tag type as an argument, such as

```
var childEl=document.createElement("div");
```

`createTextNode()` creates a DOM node representing a piece of text, commonly found nested inside heading, div, paragraph, and list item tags.

```
var txtNode=document.createTextNode("some text");
```

The DOM standard treats text nodes as separate from those representing HTML elements. They can't have styles applied to them directly and hence take up much less memory. The text represented by a text node may, however, be styled by the DOM element containing it.

Once the node, of whatever type, has been created, it must be attached to the document before it is visible in the browser window. The DOM node method `appendChild()` is used to accomplish this:

```
el.appendChild(childEl);
```

These three methods—`createElement()`, `createTextNode()`, and `appendChild()`—give us everything that we need to add new structure to a document. Having done so, however, we will generally want to style it in a suitable way, too. Let's look at how we can do this.

2.4.4 Adding styles to your document

So far, we've looked at using the DOM to manipulate the structure of a document—how one element is contained by another and so on. In effect, it allows us to reshape the structures declared in the static HTML. The DOM also provides methods for programmatically modifying the style of elements and reshaping the structures defined in the stylesheets.

Each element in a web page can have a variety of visual elements applied to it through DOM manipulation, such as position, height and width, colors, margins and borders. Modifying each attribute individually allows for very fine control, but it can be tedious. Fortunately, the web browser provides us with JavaScript bindings that allow us to exercise precision where needed through a low-level interface and to apply styling consistently and easily using CSS classes. Let's look at each of these in turn.

The className property

CSS offers a concise way of applying predefined, reusable styles to documents. When we are styling elements that we have created in code, we can also take advantage of CSS, by using a DOM node's `className` property. The following line, for example, applies the presentation rules defined by the `declared` class to a node:

```
hello.className='declared';
```

where `hello` is the reference to the DOM node. This provides an easy and compact way to assign many CSS rules at once to a node and to manage complex stylings through stylesheets.

The style property

In other situations, we may want to make a finer-grained change to a particular element's style, possibly supplementing styles already applied through CSS.

DOM nodes also contain an associative array called `style`, containing all the fine details of the node's style. As figure 2.4 illustrates, DOM node styles typically contain a large number of entries. Under the hood, assigning a `className` to the node will modify values in the `style` array.

The `style` array can be manipulated directly. After styling the items in the `empty` node, we draw a box around them; thus:

```
empty.style.border="solid green 2px";
empty.style.width="200px";
```

We could just as easily have declared a `box` class and applied it via the `className` property, but this approach can be quicker and simpler in certain circumstances, and it allows for the programmatic construction of strings. If we want to freely resize elements to pixel accuracy, for example, doing so by predefining styles for every width from 1 to 800 pixels would clearly be inefficient and cumbersome.

Using the above methods, then, we can create new DOM elements and style them. There's one more useful tool in our toolbox of content-manipulation techniques that takes a slightly different approach to programmatically writing a web page. We close this section with a look at the `innerHTML` property.

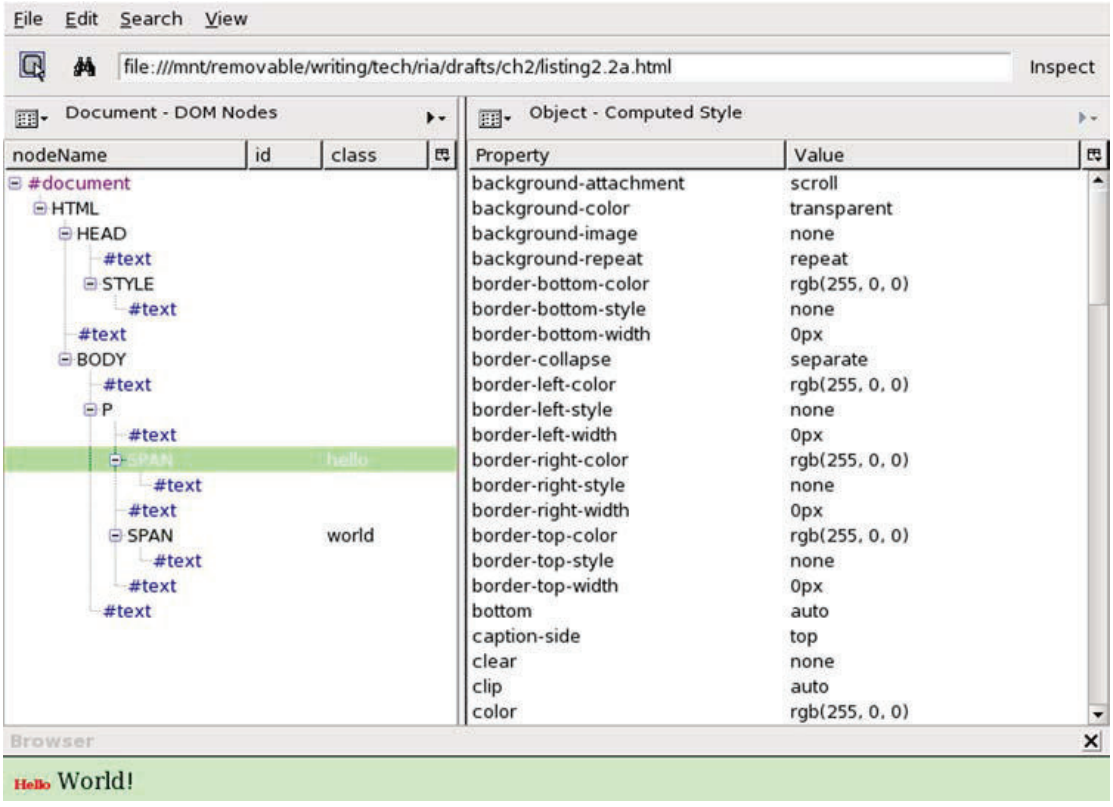

Figure 2.4 Inspecting the `style` attribute of a DOM node in the DOM Inspector. Most values will not be set explicitly by the user but will be assigned by the rendering engine itself. Note the scrollbar: we're seeing only roughly one-quarter of the full list of computed styles.

2.4.5 A shortcut: Using the innerHTML property

The methods described so far provide low-level control over the DOM API. However, `createElement()` and `appendChild()` provide a verbose API for building a document and are best suited for situations in which the document being created follows a regular structure that can be encoded as an algorithm. All popular web browsers' DOM elements also support a property named `innerHTML`, which allows arbitrary content to be assigned to an element in a very simple way. `innerHTML` is a string, representing a node's children as HTML markup. For example, we can rewrite our `addNode()` function to use `innerHTML` like this:

```
function addListItemUsingInnerHTML(el,text){
  el.innerHTML+="<div class='programmed'>"+text+"</div>";
}
```

The `<DIV>` element and the nested text node can be added in a single statement. Note also that it is appending to the property using the `+=` operator, not assigning it directly. Deleting a node using `innerHTML` would require us to extract and parse the string. `innerHTML` is less verbose and suited to relatively simple applications such as this. If a node is going to be heavily modified by an application, the DOM nodes presented earlier provide a superior mechanism.

We've now covered JavaScript, CSS, and the DOM. Together, they went under the name Dynamic HTML when first released. As we mentioned in the introduction to this chapter, Ajax uses many of the Dynamic HTML techniques, but it is new and exciting because it throws an added ingredient into the mix. In the next section, we'll look at what sets Ajax apart from DHTML—the ability to talk to the server while the user works.

2.5 Loading data asynchronously using XML technologies

While working at an application—especially a sovereign one—users will be interacting continuously with the app, as part of the workflow. In chapter 1, we discussed the importance of keeping the application responsive. If everything locks up while a lengthy background task executes, the user is interrupted. We discussed the advantages of asynchronous method calls as a way of improving UI responsiveness when executing such lengthy tasks, and we noted that, because of network latency, all calls to the server should be considered as lengthy. We also noted that under the basic HTTP request-response model, this was a bit of a nonstarter. Classical web applications rely on full-page reloads with every call to the server leading to frequent interruptions for the user.

Although we have to accept that a document request is blocked until the server returns its response, we have a number of ways of making a server request look asynchronous to users so that they can continue working. The earliest attempts at providing this background communication used IFrames. More recently, the XMLHttpRequest object has provided a cleaner and more powerful solution. We'll look at both technologies here.

2.5.1 IFrames

When DHTML arrived with versions 4 of Netscape Navigator and Microsoft Internet Explorer, it introduced flexible, programmable layout to the web page. A natural extension of the old HTML Frameset was the *IFrame*. The *I* stands for *inline*, meaning that it is part of the layout of another document, rather than sitting side by side as in a frameset. An IFrame is represented as an element in the DOM tree, meaning that we can move it about, resize it, and even hide it altogether, while the page is visible. The key breakthrough came when people started to realize that an IFrame could be styled so as to be completely invisible. This allowed it to fetch data in the background, *while the visible user experience was undisturbed*. Suddenly, there was a mechanism to contact the server asynchronously, albeit rather a hacky one. Figure 2.5 illustrates the sequence of events behind this approach.

Like other DOM elements, an IFrame can be declared in the HTML for a page or it can be programmatically generated using `document.createElement()`. In a simple case, in which we want only a single nonvisible IFrame for loading data into, we can declare it as part of the document and get a programmatic handle on it using `document.getElementById()`, as in listing 2.6.

Listing 2.6 Using an IFrame

```
<html>
<head>
<script type='text/javascript'>
window.onload=function(){
  var iframe=document.getElementById('dataFeed');
  var src='datafeeds/mydata.xml';
  loadDataAsynchronously(iframe,src);
}
function loadDataAsynchronously(iframe,src){
  //...do something amazing!!
}
</script>
</head>
<body>
<!--
...some visible content here...
```

```
-->
<iframe
  id='dataFeed'
  style='height:0px;width:0px;'
>
</iframe>
</body>
</html>
```

The IFrame has been styled as being invisible by setting its width and height to zero pixels. We could use a styling of `display:none`, but certain browsers will optimize based on this and not bother to load the document! Note also that we need to wait for the document to load before looking for the IFrame, by calling `getElementById()` in the `window.onload` handler function. Another approach is to programmatically generate the IFrames on demand, as in listing 2.7. This has the added advantage of keeping all the code related to requesting the data in one place, rather than needing to keep unique DOM node IDs in sync between the script and the HTML.

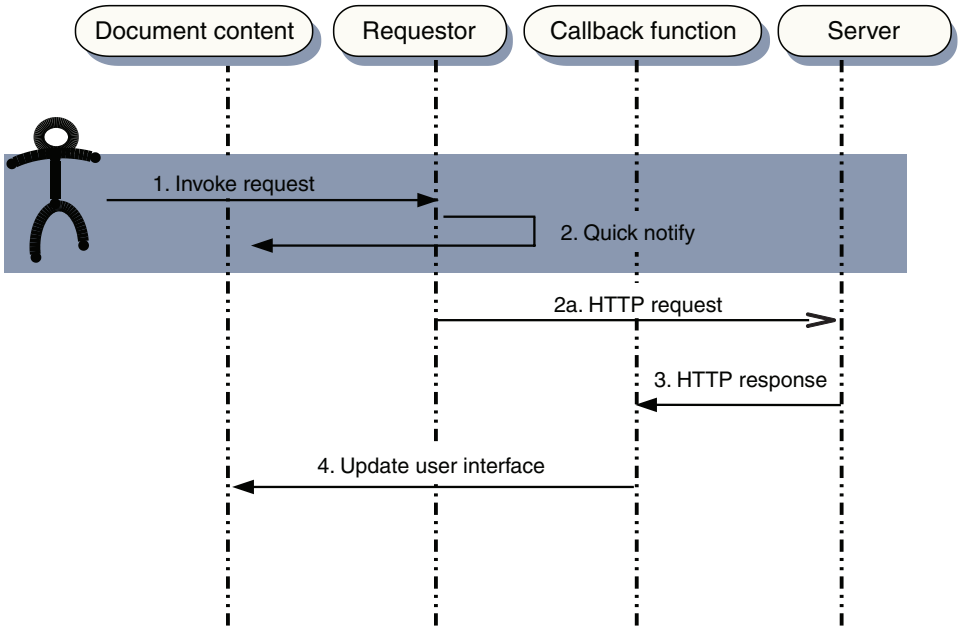

Figure 2.5 Sequence of events in an asynchronous communication in a web page. User action invokes a request from a hidden requester object (an IFrame or XMLHttpRequest object), which initiates a call to the server asynchronously. The method returns very quickly, blocking the user interface for only a short period of time, represented by the height of the shaded area. The response is parsed by a callback function, which then updates the user interface accordingly.

Listing 2.7 Creating an IFrame

```
function fetchData(){
  var iframe=document.createElement('iframe');
  iframe.className='hiddenDataFeed';
  document.body.appendChild(iframe);
  var src='datafeeds/mydata.xml';
  loadDataAsynchronously(iframe,src);
}
```

The use of `createElement()` and `appendChild()` to modify the DOM should be familiar from earlier examples. If we follow this approach rigidly, we will eventually create a large number of IFrames as the application continues to run. We need to either destroy the IFrames when we've finished with them or implement a pooling mechanism of some sort.

Design patterns, which we introduce in chapter 3, can help us to implement robust pools, queues, and other mechanisms that make a larger-scale application run smoothly, so we'll return to this topic in more depth later. In the meantime, let's turn our attention to the next set of technologies for making behind-the-scenes requests to the server.

2.5.2 XmlDocument and XMLHttpRequest objects

IFrames can be used to request data behind the scenes, as we just saw, but it is essentially a hack, repurposing something that was originally introduced to display visible content within a page. Later versions of popular web browsers introduced purpose-built objects for asynchronous data transfer, which, as we will see, offer some convenient advantages over IFrames.

The XmlDocument and XMLHttpRequest objects are nonstandard extensions to the web browser DOM that happen to be supported by the majority of browsers. They streamline the business of making asynchronous calls considerably, because they are explicitly designed for fetching data in the background. Both objects originated as Microsoft-specific ActiveX components that were available as JavaScript objects in the Internet Explorer browser. Other browsers have since implemented native objects with similar functionality and API calls. Both perform similar functions, but the XMLHttpRequest provides more fine-grained control over the request. We will use that throughout most of this book, but mention XmlDocument briefly here in case you come across it and wonder how it differs from XMLHttpRequest. Listing 2.8 shows a simple function body that creates an XmlDocument object.

Listing 2.8 `getXmlDocument()` function

```
function getXMLDocument(){
  var xDoc=null;
  if (document.implementation
    && document.implementation.createDocument){
    xDoc=document.implementation
     .createDocument("","",null);    <-- Mozilla/Safari
  }else if (typeof ActiveXObject != "undefined"){
    var msXmlAx==null;
    try{
      msXmlAx=new ActiveXObject
       ("Msxml2.DOMDocument");    <-- Newer Internet Explorer
    }catch (e){
      msXmlAx=new ActiveXObject
       ("Msxml.DOMDocument");    <-- Older Internet Explorer

    }
    xDoc=msXmlAx;
  }
  if (xDoc==null || typeof xDoc.load=="undefined"){
    xDoc=null;
  }
  return xDoc;
}
```

The function will return an XmlDocument object with an identical API under most modern browsers. The ways of creating the document differ considerably, though.

The code checks whether the document object supports the implementation property needed to create a native XmlDocument object (which it will find in recent Mozilla and Safari browsers). If it fails to find one, it will fall back on ActiveX objects, testing to see if they are supported or unsupported (which is true only in Microsoft browsers) and, if so, trying to locate an appropriate object. The script shows a preference for the more recent MSXML version 2 libraries.

NOTE It is possible to ask the browser for vendor and version number information, and it is common practice to use this information to branch the code based on browser type. Such practice is, in our opinion, prone to error, as it cannot anticipate future versions or makes of browser and can exclude browsers that are capable of executing a script. In our `getXmlDocument()` function, we don't try to guess the version of the browser but ask directly whether certain objects are available. This approach, known as *object detection*, stands a better chance of working in future versions of browsers, or in unusual browsers that we haven't explicitly tested, and is generally more robust.

Listing 2.9 follows a similar but slightly simpler route for the XMLHttpRequest object.

Listing 2.9 `getXmlHttpRequest()` function

```
function getXMLHTTPRequest() {
  var xRequest=null;
  if (window.XMLHttpRequest) {
    xRequest=new XMLHttpRequest();    <-- Mozilla/Safari
  }else if (typeof ActiveXObject != "undefined"){
    xRequest=new ActiveXObject
     ("Microsoft.XMLHTTP");    <-- Internet Explorer
  }
  return xRequest;
}
```

Again, we use object detection to test for support of the native XMLHttpRequest object and, failing that, for support for ActiveX. In a browser that supports neither, we will simply return `null` for the moment. We'll look at gracefully handling failure conditions in more detail in chapter 6.

So, we can create an object that will send requests to the server for us. What do we do now that we have it?

2.5.3 Sending a request to the server

Sending a request to the server from an XMLHttpRequest object is pretty straightforward. All we need to do is pass it the URL of the server page that will generate the data for us. Here's how it's done:

```
function sendRequest(url,params,HttpMethod){
  if (!HttpMethod){
    HttpMethod="POST";
  }
  var req=getXMLHTTPRequest();
  if (req){
    req.open(HttpMethod,url,true);
    req.setRequestHeader
         ("Content-Type",
          "application/x-www-form-urlencoded");
    req.send(params);
  }
}
```

XMLHttpRequest supports a broad range of HTTP calling semantics, including optional querystring parameters for dynamically generated pages. (You may know these as CGI parameters, Forms arguments, or ServletRequest parameters,

depending on your server development background.) Let's quickly review the basics of HTTP before seeing how our request object supports it.

HTTP—A quick primer

HTTP is such a ubiquitous feature of the Internet that we commonly ignore it. When writing classic web applications, the closest that we generally get to the HTTP protocol is to define a hyperlink and possibly set the `method` attribute on a form. Ajax, in contrast, opens up the low-level details of the protocol for us to play with, allowing us to do a few surprising things.

An HTTP transaction between a browser and a web server consists of a request by the browser, followed by a response from the server (with some exceptionally clever, mind-blowingly cool code written by us web developers happening in between, of course). Both request and response are essentially streams of text, which the client and server interpret as a series of headers followed by a body. Think of the headers as lines of an address written on an envelope and the body as the letter inside. The headers simply instruct the receiving party what to do with the letter contents.

An HTTP request is mostly composed of headers, with the body possibly containing some data or parameters. The response typically contains the HTML markup for the returning page. A useful utility for Mozilla browsers called LiveHTTPHeaders (see the Resources section at the end of this chapter and appendix A) lets us watch the headers from requests and responses as the browser works. Let's fetch the Google home page and see what happens under the hood.

The first request that we send contains the following headers:

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0
  (Windows; U; Windows NT 5.0; en-US; rv:1.7)
  Gecko/20040803 Firefox/0.9.3
Accept: text/xml,application/xml,
  application/xhtml+xml,text/html;q=0.9,
  text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PREF=ID=cabd38877dc0b6a1:TM=1116601572
  :LM=1116601572:S=GD3SsQk3v0adtSBP
```

The first line tells us which HTTP method we are using. Most web developers are familiar with `GET`, which is used to fetch documents, and `POST`, used to submit

HTML forms. The World Wide Web Consortium (W3C) spec includes a few other common methods, including `HEAD`, which fetches the headers only for a file; `PUT`, for uploading documents to the server; and `DELETE`, for removing documents. Subsequent headers do a lot of negotiation, with the client telling the server what content types, character sets, and so on it can understand. Because I've visited Google before, it also sends a *cookie*, a short message telling Google who I am.

The response headers, shown here, also contain quite a lot of information:

```
HTTP/1.x 302 Found
Location: http://www.google.co.uk/cxfer?c=PREF%3D:
  TM%3D1116601572:S%3DzFxPsBpXhZzknVMF&prev=/
Set-Cookie: PREF=ID=cabd38877dc0b6a1:CR=1:TM=1116601572:
  LM=1116943140:S=fRfhD-u49xp9UE18;
  expires=Sun, 17-Jan-2038 19:14:07 GMT;
  path=/; domain=.google.com
Content-Type: text/html
Server: GWS/2.1
Transfer-Encoding: chunked
Content-Encoding: gzip
Date: Tue, 24 May 2005 17:59:00 GMT
Cache-Control: private, x-gzip-ok=""
```

The first line indicates the status of the response. A `302` response indicates a redirection to a different page. In addition, another cookie is passed back for this session. The content type of the response (aka MIME type) is also declared. A further request is made on the strength of the redirect instruction, resulting in a second response with the following headers:

```
HTTP/1.x 200 OK
Cache-Control: private
Content-Type: text/html
Content-Encoding: gzip
Server: GWS/2.1
Content-Length: 1196
Date: Tue, 24 May 2005 17:59:00 GMT
```

Status code `200` indicates success, and the Google home page will be attached to the body of this response for display. The content-type header tells the browser that it is `html`.

Our `sendRequest()` method is constructed so that the second and third parameters, which we probably won't need most of the time, are optional, defaulting to using POST to retrieve the resource with no parameters passed in the request body.

The code in this listing sets the request in motion and will return control to us immediately, while the network and the server take their own sweet time.

This is good for responsiveness, but how do we find out when the request has completed?

2.5.4 Using callback functions to monitor the request

The second part of the equation for handling asynchronous communications is setting up a reentry point in your code for picking up the results of the call once it has finished. This is generally implemented by assigning a callback function, that is, a piece of code that will be invoked when the results are ready, at some unspecified point in the future. The `window.onload` function that we saw in listing 2.9 is a callback function.

Callback functions fit the event-driven programming approach used in most modern UI toolkits—keyboard presses, mouse clicks, and so on will occur at unpredictable points in the future, too, and the programmer anticipates them by writing a function to handle them when they do occur. When coding UI events in JavaScript, we assign functions to the `onkeypress`, `onmouseover`, and similarly named properties of an object. When coding server request callbacks, we encounter similar properties called `onload` and `onreadystatechange`.

Both Internet Explorer and Mozilla support the `onreadystatechange` callback, so we'll use that. (Mozilla also supports `onload`, which is a bit more straightforward, but it doesn't give us any information that `onreadystatechange` doesn't.) A simple callback handler is demonstrated in listing 2.10.

Listing 2.10 Using a callback handler

```
var READY_STATE_UNINITIALIZED=0;
var READY_STATE_LOADING=1;
var READY_STATE_LOADED=2;
var READY_STATE_INTERACTIVE=3;
var READY_STATE_COMPLETE=4;
var req;
function sendRequest(url,params,HttpMethod){
  if (!HttpMethod){
    HttpMethod="GET";
  }
  req=getXMLHTTPRequest();
  if (req){
    req.onreadystatechange=onReadyStateChange;
    req.open(HttpMethod,url,true);
    req.setRequestHeader
         ("Content-Type", "application/x-www-form-urlencoded");
    req.send(params);
  }
}
```

```
function onReadyStateChange(){
  var ready=req.readyState;
  var data=null;
  if (ready==READY_STATE_COMPLETE){
    data=req.responseText;
  }else{
    data="loading...["+ready+"]";
  }
  //... do something with the data...
}
```

First, we alter our `sendRequest()` function to tell the request object what its callback handler is, before we send it off. Second, we define the handler function, which we have rather unimaginatively called `onReadyStateChange()`.

`readyState` can take a range of numerical values. We've assigned descriptively named variables to each here, to make our code easier to read. At the moment, the code is only interested in checking for the value `4`, corresponding to completion of the request.

Note that we declare the request object as a global variable. Right now, this keeps things simple while we address the mechanics of the `XMLHttpRequest` object, but it could get us into trouble if we were trying to fire off several requests simultaneously. We'll show you how to get around this issue in section 3.1. Let's put the pieces together now, to see how to handle a request end to end.

2.5.5 The full lifecycle

We now have enough information to bring together the complete lifecycle of loading a document, as illustrated in listing 2.11. We instantiate the XMLHttpRequest object, tell it to load a document, and then monitor that load process asynchronously using callback handlers. In the simple example, we define a DOM node called `console`, to which we can output status information, in order to get a written record of the download process.

Listing 2.11 Full end-to-end example of document loading using XMLHttpRequest

```
<html>
<head>
<script type='text/javascript'>
var req=null;
var console=null;
var READY_STATE_UNINITIALIZED=0;
var READY_STATE_LOADING=1;
var READY_STATE_LOADED=2;
```

```
var READY_STATE_INTERACTIVE=3;
var READY_STATE_COMPLETE=4;
function sendRequest(url,params,HttpMethod){
  if (!HttpMethod){
    HttpMethod="GET";
  }
  req=initXMLHTTPRequest();
  if (req){
    req.onreadystatechange=onReadyState;
    req.open(HttpMethod,url,true);
    req.setRequestHeader
         ("Content-Type", "application/x-www-form-urlencoded");
    req.send(params);
  }
}
function initXMLHTTPRequest(){
  var xRequest=null;
  if (window.XMLHttpRequest){
    xRequest=new XMLHttpRequest();
  } else if (window.ActiveXObject){
    xRequest=new ActiveXObject
     ("Microsoft.XMLHTTP");
  }
  return xRequest;
}
function onReadyState(){
  var ready=req.readyState;
  var data=null;
  if (ready==READY_STATE_COMPLETE){
    data=req.responseText;
  }else{
    data="loading...["+ready+"]";
  }
  toConsole(data);
}
function toConsole(data){
  if (console!=null){
    var newline=document.createElement("div");
    console.appendChild(newline);
    var txt=document.createTextNode(data);
    newline.appendChild(txt);
  }
}
window.onload=function(){
  console=document.getElementById('console');
  sendRequest("data.txt");
}
</script>
</head>
<body>
<div id='console'></div>
```

Initialize request object

Define callback handler

Check readyState

Read response data

```
</body>
</html>
```

Let's look at the output of this program in Microsoft Internet Explorer and Mozilla Firefox, respectively. Note that the sequence of readyStates is different, but the end result is the same. The important point is that the fine details of the readyState shouldn't be relied on in a cross-browser program (or indeed, one that is expected to support multiple versions of the same browser). Here is the output in Microsoft Internet Explorer:

```
loading...[1]
loading...[1]
loading...[3]
Here is some text from the server!
```

Each line of output represents a separate invocation of our callback handler. It is called twice during the loading state, as each chunk of data is loaded up, and then again in the interactive state, at which point control would be returned to the UI under a synchronous request. The final callback is in the completed state, and the text from the response can be displayed.

Now let's look at the output in Mozilla Firefox version 1.0:

```
loading...[1]
loading...[1]
loading...[2]
loading...[3]
Here is some text from the server!
```

The sequence of callbacks is similar to Internet Explorer, with an additional callback in the loaded readyState, with value of `2`.

In this example, we used the `responseText` property of the XMLHttpRequest object to retrieve the response as a text string. This is useful for simple data, but if we require a larger structured collection of data to be returned to us, then we can use the `responseXML` property. If the response has been allocated the correct MIME type of text/xml, then this will return a DOM document that we can interrogate using the DOM properties and functions such as `getElementById()` and `childNodes` that we encountered in section 2.4.1.

These, then, are the building blocks of Ajax. Each brings something useful to the party, but a lot of the power of Ajax comes from the way in which the parts combine into a whole. In the following section, we'll round out our introduction to the technologies with a look at this bigger picture.

2.6 What sets Ajax apart

While CSS, DOM, asynchronous requests, and JavaScript are all necessary components of Ajax, it is quite possible to use all of them *without* doing Ajax, at least in the sense that we are describing it in this book.

We already discussed the differences between the classic web application and its Ajax counterpart in chapter 1; let's recap briefly here. In a classic web application, the user workflow is defined by code on the server, and the user moves from one page to another, punctuated by the reloading of the entire page. During these reloads, the user cannot continue with his work. In an Ajax application, the workflow is at least partly defined by the client application, and contact is made with the server in the background while the user gets on with his work.

In between these extremes are many shades of gray. A web application may deliver a series of discrete pages following the classic approach, in which each page cleverly uses CSS, DOM, JavaScript, and asynchronous request objects to smooth out the user's interaction with the page, followed by an abrupt halt in productivity while the next page loads. A JavaScript application may present the user with page-like pop-up windows that behave like classic web pages at certain points in the flow. The web browser is a flexible and forgiving environment, and Ajax and non-Ajax functionality can be intermingled in the same application.

What sets Ajax apart is not the technologies that it employs but the interaction model that it enables through the use of those technologies. The web-based interaction model to which we are accustomed is not suited to sovereign applications, and new possibilities begin to emerge as we break away from that interaction model.

There are at least two levels at which Ajax can be used—and several positions between these as we let go of the classic page-based approach. The simplest strategy is to develop Ajax-based widgets that are largely self-contained and that can be added to a web page with a few imports and script statements. Stock tickers, interactive calendars, and chat windows might be typical of this sort of widget. Islands of application-like functionality are embedded into a document-like web page (figure 2.6). Most of Google's current forays into Ajax (see section 1.3) fit this model. The drop-down box of Google Suggest and the map widget in Google Maps are both interactive elements embedded into a page.

If we want to adopt Ajax more adventurously, we can turn this model inside out, developing a host application in which application-like and document-like fragments can reside (figure 2.7). This approach is more analogous to a desktop application, or even a window manager or desktop environment. Google's GMail

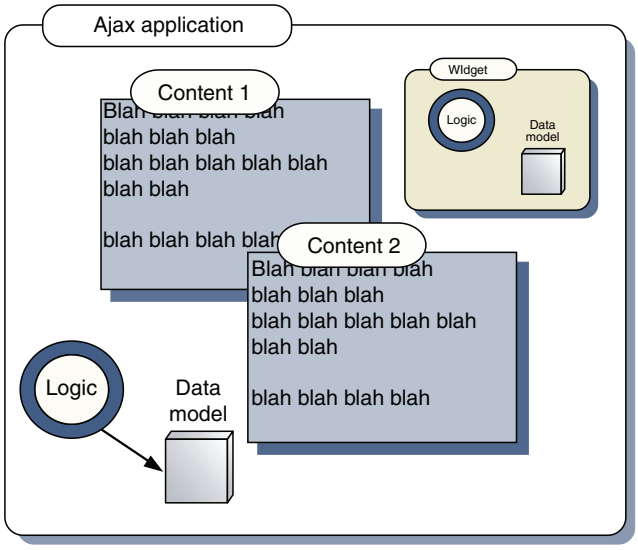

Figure 2.6
A simple Ajax application will still work like a web page, with islands of interactive functionality embedded in the page.

fits this model, with individual messages rendering as documents within an interactive, application-like superstructure.

In some ways, learning the technologies is the easy part. The interesting challenge in developing with Ajax is in learning how to use them together. We are accustomed to thinking of web applications as storyboards, and we shunt the user from one page to another following a predetermined script. With application-like functionality in our web application, we can provide the user with a more fine-grained handle on the business domain, which can enable a more free-form problem-solving approach to his work.

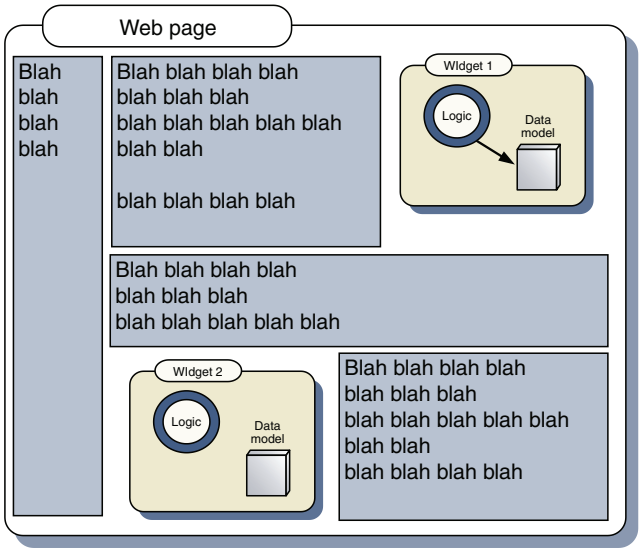

Figure 2.7
In a more complex Ajax application, the entire application is an interactive system, into which islands of document-like content may be loaded or programmatically declared.

In order to gain the benefits of this greater flexibility, we have to question a lot of our coding habits. Is an HTML form the only way for a user to input information? Should we declare all our user interfaces as HTML? Can we contact the server in response to user interactions such as key presses and mouse movements, as well as the conventional mouse click? In the fast-paced world of information technology, we place a large emphasis on learning new skills, but unlearning old habits can be at least as important.

2.7 Summary

In this chapter, we've introduced the four technical pillars of Ajax.

JavaScript is a powerful general-purpose programming language with a bad reputation for generating pop-up windows, back-button hacks, and image rollovers. Appendix B contains a more detailed description of some of the features of the language, but from the examples here, you should be able to get a feel for how it can be used to genuinely enhance usability.

CSS and the DOM complement one another in providing a clear programmatic view of the user interface that we're working with, while keeping the structure separate from the visual styling. A clean document structure makes programmatic manipulation of a document much simpler, and maintaining a separation of responsibilities is important in developing larger Ajax applications, as we'll see in chapters 3 and 4.

We've shown how to work with the XMLHttpRequest object and with the older XmlDocument and IFrame. A lot of the current hype around Ajax praises XMLHttpRequest as the fashionable way to talk to the server, but the IFrame offers a different set of functionality that can be exactly what we need at times. Knowing about both enriches your toolkit. In this chapter, we introduced these techniques and provided some examples. In chapter 5, we will discuss client/server communications in more detail.

Finally, we looked at the way the technological pillars of Ajax can be combined to create something greater than the sum of its parts. While Ajax can be used in small doses to add compelling widgets to otherwise static web pages, it can also be applied more boldly to create a complete user interface within which islands of static content can be contained. Making this leap from the sidelines to center stage will require a lot of JavaScript code, however, and that code will be required to run without fail for longer periods, too. This will require us to approach our code differently and look at such issues as reliability, maintainability, and flexibility. In the next chapter, we look at ways of introducing order into a large-scale Ajax codebase.

2.8 Resources

For a deeper understanding of Cascading Style Sheets, we recommend the CSS Zen Garden (www.csszengarden.com/), a site that restyles itself in a myriad of ways using nothing but CSS.

Eric Meyer has also written extensively on CSS; visit his website at www.meyerweb.com/eric/css/. Blooberry (www.blooberry.com) is another excellent website for CSS information.

Early Ajax solutions using IFrames are described at http://developer.apple.com/internet/webcontent/iframe.html.

The LiveHttpHeaders extension for Mozilla can be found at http://livehttpheaders.mozdev.org/

Danny Goodman's books on JavaScript are an essential reference for DOM programming, and cover the browser environments in great detail: *Dynamic* HTML*: The Definitive Reference* (O'Reilly 2002) and *JavaScript Bible* (John Wiley 2004).

The W3Schools website contains some interactive tutorials on JavaScript, for those who like to learn by doing (www.w3schools.com/js/js_examples_3. asp).